

Programación de

SISTEMAS EMBEBIDOS

en C

Gustavo Galeano



Alfaomega



INDICE

Introducción

- 1.1 ¿Qué es un Sistema Embebido?
- 1.2.1 ¿Qué es un compilador? ¿Qué es un interpretador?
- 1.2.2 Estructura y pasadas de un compilador
- 1.2.3 Tiempo de compilación vs tiempo de ejecución
- 1.3 ¿De dónde viene el ANSI C?
- 1.3.1 Forma general de un proyecto en C
- 1.4 Periféricos más comunes en Sistemas Embebidos
 - 1.4.1 Puertos de entrada/salida y Función KBI
 - 1.4.2 Conversor analógico a digital (ADC)
 - 1.4.3 Computador operando apropiadamente (COP)
 - 1.4.4 Detector de bajo nivel de voltaje (LVI)
 - 1.4.5 Temporizador (TIMER)
 - 1.4.6 Comunicación serial asincrónica (SCI)
 - 1.4.7 Comunicación serial sincrónica (SPI)
 - 1.4.8 Comunicación serial I2C
- 1.5.1 El Concepto de Interrupción
- 1.5.2 ¿Cómo trabaja el procesador ante una interrupción?
- 1.6 Cambio de contexto
- 1.7 Latencia de interrupción
- 1.8 Zonas críticas de software
 - 1.9.1 Herramientas para Microchip™
 - 1.9.2 Herramientas para Renesas™
 - 1.9.3 Herramientas para Texas Instruments
 - 1.9.4 Herramientas para Freescale™
 - 1.9.5 Herramientas utilizadas en los ejemplos
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 2.1 Arquitectura RISC (Harvard) Microchip de 8 bits
 - 2.1.1 Componentes básicos de la arquitectura Microchip™
 - 2.1.2 Modos de direccionamiento Microchip™
 - 2.1.3 Mapa de memoria Microchip™ del PIC16F877A
 - 2.1.4 Características del microcontrolador PIC16F877A
 - 2.1.5 Fuentes de interrupción del microcontrolador PIC16F877A
- 2.2 Arquitectura CISC (Von Newman) Freescale™ de 8 bits
 - 2.2.1 Modelo de programación Freescale™ HC(S) 08
 - 2.2.2 Modos de direccionamiento Freescale™ HC(S) 08
 - 2.2.3 Mapa de memoria del microcontrolador AP16A

- 2.2.4 Características del microcontrolador AP16A
- 2.2.5 Fuentes de interrupción del microcontrolador AP16A
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 3.1 El manejador de Proyectos (IDE)
- 3.2 Introducción al compilador CCS para Microchip™
 - 3.2.1 Creación de proyectos embebidos usando el ambiente CCS
 - 3.2.2 Familiarización con el IDE del CCS
- 3.3 Introducción al compilador Codewarrior® de Freescale™
 - 3.3.1 Creación de proyectos embebidos en C usando Codewarrior®
 - 3.3.2 Familiarización con el IDE de Codewarrior®
- 3.4 El primer programa embebido en C usando Codewarrior®
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 4.1 Ventajas y desventajas del C usado en Sistemas Embebidos
- 4.2 Programación embebida vs programación para PC
- 4.3 Constantes y variables
 - 4.3.1 Constantes
 - 4.3.2 Variables
 - 4.4.1 char en Codewarrior ® (o int8 en CCS)
 - 4.4.2 int en Codewarrior ® (o int16 para CCS)
 - 4.4.3 long en Codewarrior ® (o int32 para CCS)
 - 4.4.4 float (en Codewarrior ® y en CCS)
 - 4.4.5. Double en Codewarrior ® (no soportado en CCS)
 - 4.5 ¿Cómo elegir un tipo de variable en un sistema embebido?
- 4.6 Modificadores a tipos de datos comunes en Sistemas Embebidos
 - 4.6.1 El modificador “unsigned”
 - 4.6.2 El modificador “signed”
 - 4.6.3 El modificador “volatile”
 - 4.6.4 El modificador “near”
 - 4.6.5 El modificador “far”
 - 4.6.6 El modificador “register”
 - 4.6.7 El modificador “const”
 - 4.6.8 El modificador “static”
 - 4.6.9 El modificador “extern”
- 4.7 Moldes o “Casting” para variables y constantes
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 5.1 Directivas más comunes del preprocesador
 - 5.1.1 Macro o equivalencia #define
 - 5.1.2 Inclusión de archivo de cabecera #include
 - 5.1.3 Notificación de error al compilar #error
 - 5.1.4 Notificación de precaución al compilar #warning
 - 5.1.5 Directiva #line
 - 5.1.6 Compilación condicional #if, #elif, #else, #endif, #ifdef y #ifndef
 - 5.1.7 Directiva #undef
 - 5.1.8 Directiva #pragma
- 5.2 Medida en bytes de expresiones “sizeof”
- 5.3 Definiciones de tipo “typedef”
- 5.4 Operadores aritméticos
- 5.5 Operadores relacionales
 - 5.5.1 Operadores de Comparación cuantitativa: <, >, <=, >=
 - 5.5.2 Operador de igualdad: ==, !=
- 5.6 Operadores lógicos booleanos
 - 5.6.1 El operador || (OR)
 - 5.6.2 El operador && (AND)
 - 5.6.3 El operador ! (NOT)
- 5.7 Operadores orientados a BIT
 - 5.7.1 El operador | (OR Bit a Bit)
 - 5.7.2 El operador & (AND bit a bit)
 - 5.7.3 El operador ~ (NOT bit a bit)
 - 5.7.4 El operador ^ (OR exclusiva o XOR)
- 5.8 Los operadores >> y << (Desplazamiento)
- 5.9 Los operadores de APUNTADOR & y *
- 5.9.1 Declaración de variables tipo apuntador
- 5.10 Precedencia y asociatividad
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 6.1 Funciones en C para Sistemas Embebidos
 - 6.1.1 Prototipo de una función
 - 6.1.2 El concepto de paso de argumentos a función
 - 6.1.3 Paso de argumentos por valor
 - 6.1.4 Archivos de cabecera (.H)
 - 6.1.5 Paso de argumentos por referencia
- 6.2 Sentencias de control
 - 6.2.1 La sentencia “if... else”

6.2.2 La sentencia “do... while”
6.2.3 La sentencia “while”
6.2.4 La sentencia “for”
6.2.5 La sentencia “switch”
6.2.6 La sentencia “break”
6.2.7 La sentencia “continue”
6.2.8 La sentencia “goto”
6.3 Mezcla de C con lenguaje ensamblador en Sistemas Embebidos
6.4 Recursividad en Sistemas Embebidos
6.5 Arreglos (Arrays) de datos
6.5.1 Arreglos unidimensionales
6.5.2 Cadenas de datos “strings”
6.5.3 Arreglos multidimensionales
6.5.4 Estructuras “struct” y uniones “union”
6.5.5 Estructuras de bits
6.6 Apuntadores a funciones
6.7 Manejo de interrupciones en Sistemas Embebidos desde C
6.7.1 Configuración de interrupciones en Codewarrior ®
6.8 Convenciones útiles de programación embebida en C
6.8.1 Sobre la distribución del código fuente
6.8.2 Sobre los nombres de funciones
6.8.3 Sobre las variables y apuntadores
6.8.4 Sobre las constantes
6.8.5 Sobre los paréntesis y corchetes
Resumen del capítulo
Preguntas del capítulo

Introducción

7.1 ¿Qué es una librería ANSI C?
7.2 Librería matemática <math.h>
7.3 Librería estándar <stdlib.h>
7.4 Librería estándar de entrada/salida <stdio.>
7.5 Librería de manejo de cadenas <string.h>
7.6 Librería de tipos <ctype.h>
7.7 Librería de manejo de tiempo <time.h>
Resumen del capítulo
Preguntas del capítulo

Introducción

8.1 La ecuación de consumo de energía
8.2 Modo “Wait”
8.3 Modo “Stop”

- 8.3.1 Modo “Stop3”
- 8.3.2 Modo “Stop2”
- 8.3.3 Modo “Stop1”
- 8.4 Otros modos de bajo consumo
 - 8.4.1 Modo “Low Power Run” (LPrun)
 - 8.4.2 Modo “Low Power Wait” (LPwait)
- 8.5 Consideraciones en el diseño de un Sistema Embbebido
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 9.1 “Proccess or Exper tTM”
- 9.2 El Concepto del “bean”
- 9.3 Librería de “beans”
- 9.4 Creación de proyectos usando el “Processor Expert™”
 - 9.5.1 La ventana “bean inspector”
 - 9.5.2 La ventana “bean selector”
 - 9.5.3 La ventana “Target CPU”
- 9.6 Creación de beans en “Processor Expert™”
 - 9.6.1 Creación de un bean plantilla
 - 9.6.2 Uso del Bean Wizard™
- 9.7 Consideraciones sobre el uso del “Processor Expert™”
- Resumen del capítulo
- Preguntas del capítulo

Introducción

- 10.1 ¿Qué es un sistema operativo de tiempo real?
- 10.2.1 Tareas (Task ó Thread)
- 10.2.2 Recursos (resources)
- 10.2.3 Eventos (events)
- 10.2.4 Semáforos (semaphores)
- 10.2.5 Mensajes (message mailbox)
- 10.2.6 Bloques de Memoria (buffers)
- 10.2.7 Reloj y Timers (Clock Tick & Timers)
- 10.2.8 Kernel
- 10.3 Sistema de Loop consecutivo/interrupción
 - 10.3.1 Diseño del Loop principal
 - 10.3.2 Archivo de cabecera TareaX.H.
 - 10.3.3 Archivo código fuente TareaX.C.
 - 10.3.4 Manejo de prioridades
- 10.4 Sistema RTOS uC/OS-II de Micrium.
 - 10.4.1 Consideración para la implementación del uC/OS-II.

10.4.2 Manejo de las secciones críticas de software en uC/OS-II.

10.4.3 Portado del uC/OS-II para el AP16A

Resumen del capítulo

Preguntas del capítulo

Anexo_1

Anexo_2

Glosario

Bibliografía

INTRODUCCIÓN

El capítulo está orientado a definir los conceptos sobre programación general que servirán durante el completo recorrido del texto; inicialmente se define lo que es un sistema embebido, mostrando el modelo expandido hasta llegar al popular modelo de una sola pastilla (*single chip*).

Se aclara el concepto de compilador con respecto al de interpretador y se muestran los pasos típicos que un compilador tiene que realizar antes de generar el programa ejecutable en el chip, dejando muy claro los tiempos que tardan los procesadores involucrados en el diseño: el que realiza la compilación (tiempo de compilación) y el que ejecuta el código generado por el primero (tiempo de ejecución).

Un poco de historia sobre los orígenes del C y el porqué ha llegado a convertirse en el lenguaje más popular de programación, tanto para programadores de alto nivel en computadoras comerciales como para programadores de microcontroladores de bajo costo. Se muestra de forma general la estructura de un código desarrollado en lenguaje C y se familiariza al programador, que viene del lenguaje de ensamblador, con las palabras reservadas, los operadores y los signos propios de un código en lenguaje C.

Se abordan algunos de los periféricos existentes en los microcontroladores del mercado, con los cuales se realizarán prácticas reales en capítulos posteriores, como son los dispositivos de entrada y salida (I/O), el convertidor analógico a digital (ADC), el perro guardián (COP), el detector de bajo nivel de voltaje (LVI), el temporizador (TIMER), y las comunicaciones seriales asincrónicas (SCI) y sincrónicas (SPI, I2C).

Se dedica un buen espacio a un tema con el que siempre tienen que ver los sistemas embebidos, como lo es el de las interrupciones, parte fundamental y crítica en cualquier diseño basado en microcontroladores. Este concepto será aclarado con un ejemplo de la vida cotidiana y luego puesto en el contexto propio de un procesador.

Al final se describen las herramientas de desarrollo de cuatro de las marcas más populares de fabricantes de microcontroladores como son las de **Microchip**, **Renesas**, **Texas Instruments** y **Freescale** y la posibilidad de tener una herramienta de diseño propia a muy bajo costo, para ejercitarse de forma práctica todos los ejemplos del libro.

1.1 ¿QUÉ ES UN SISTEMA EMBEBIDO?

Se conoce como **sistema embebido** a un circuito electrónico computarizado que está diseñado para cumplir una labor específica en un producto.

La inteligencia artificial, secuencias y algoritmos de un sistema embebido, están residentes en la memoria de una pequeña computadora denominada microcontrolador.

A diferencia de los sistemas computacionales de oficina y *laptops*, estos sistemas solucionan un problema específico y están dispersos en todos los ambientes posibles de la vida cotidiana. Es



Los sistemas
embebidos
controlan un

común encontrar sistemas embebidos en los vehículos; por ejemplo, controlando el sistema de inyección de combustible, en los sistemas de frenado ABS (*Anti-lock Braking Systems*), en el control de espejos, sistemas de protección contra impacto (*Airbag*), alarmas contra robo, sistemas de ubicación, entre otros. También en los electrodomésticos de uso diario: controlando la temperatura en refrigeradores, estufas, hornos microondas y planchas; el motor de licuadoras, lavadoras de ropa, lavaplatos, aspiradoras y juguetes; en los equipos celulares, agendas de bolsillo, PDA, cajeros automáticos, cámaras fotográficas, reproductores de música (MP3) y video, equipo de gimnasio, equipo médico, y en general, en una gran cantidad de dispositivos de uso diario (*ver Gráfico 1.1*).

Se sabe que en general, un consumidor promedio interactúa con alrededor de 400 microcontroladores por día; este número tiende a crecer significativamente para los próximos años, considerando que los procesadores son cada vez mas pequeños, consumen menos energía y el precio es menor gracias a la economía de escala aplicada en su fabricación, aspectos que ayudan a reemplazar en mayor proporción los sistemas lógicos, los equipos electromecánicos y en el futuro, se podrán incorporar en los equipos desechables.

sinnúmero de objetos cotidianos, como celulares, agendas de bolsillo, cámaras fotográficas o sistemas de inyección de combustible, entre otros.



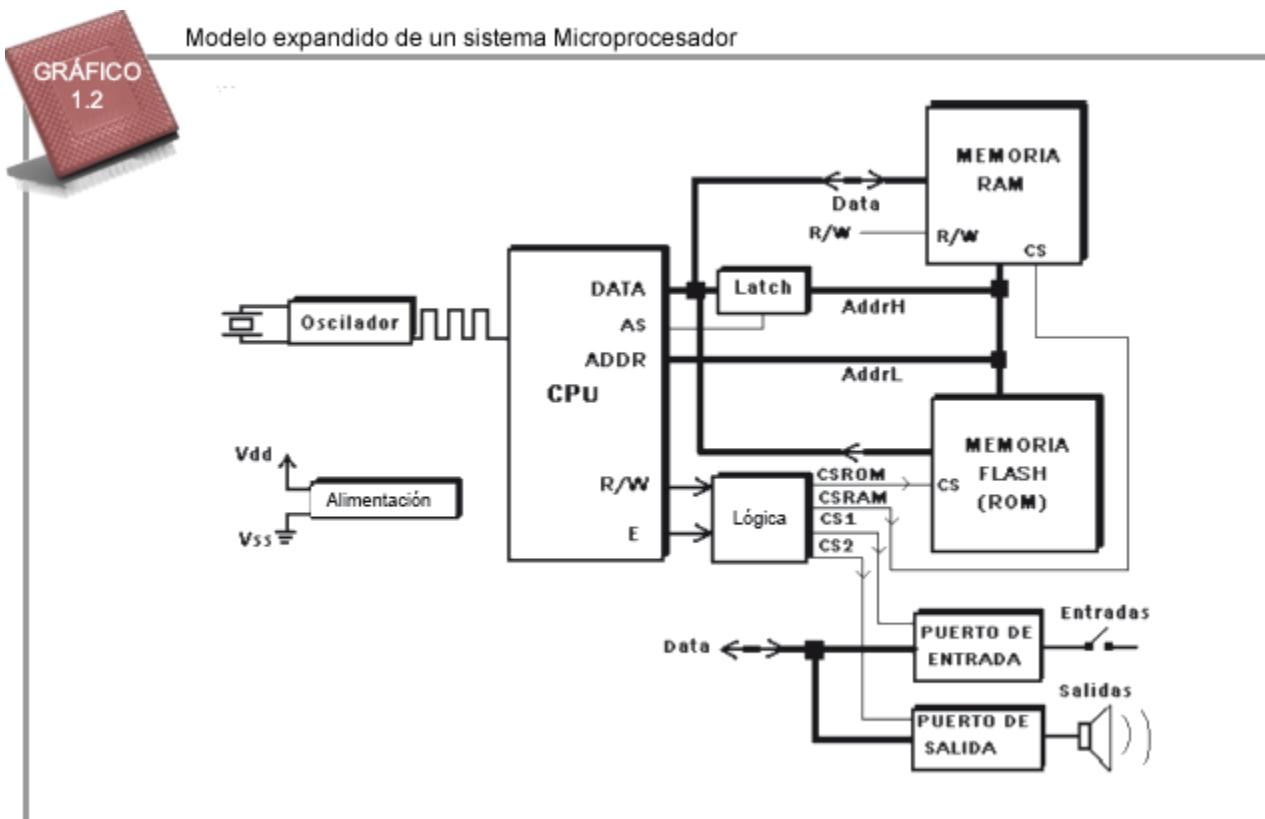
La programación
embebida

La programación embebida, como es el caso de este libro, va siempre orientada a aplicaciones portátiles o compactas, alimentadas por batería o por una fuente de poder de baja capacidad de corriente (por cuestiones de espacio), menor disipación de calor y muy económica.

permite desarrollar instrucciones precisas para microcontroladores que cumplen funciones específicas; por ejemplo, controlar el ciclo de trabajo en una lavadora.

Lo anterior apunta a tener una aplicación final de tamaño reducido y de bajo costo; sin embargo, es propio de los sistemas embebidos su robustez. Esta característica se debe al gran rango de aplicaciones y ambientes que cubren: industriales, de consumo, automotriz, electrodomésticos, donde el equipo final puede estar sometido a situaciones muy exigentes como pueden ser el polvo, la humedad, la vibración, rotaciones de alta velocidad, situaciones extremas de presión y/o temperatura.

El número de aplicaciones y de ambientes soportados por los sistemas embebidos crece cada vez más, y una de las principales razones fue la llegada de los procesadores de una sola pastilla (microcontroladores *single-chip*), en los cuales una gran parte de la electrónica está incorporada y permite reducciones de tamaño, menor consumo y facilidades de producción.



Descripción de señales:

AS: Señal de Address Strobe.

R/W: Señal de Read/Write.

E: Señal de Sincronización.

CSROM: Chip Select de Memoria de programa.

CSRAM: Chip Select de Memoria de datos.

Data: Bus de Datos bidireccional.

ADDR: Bus de Direcciones (AddrH: AddrL).

Oscilador: Señal de reloj.

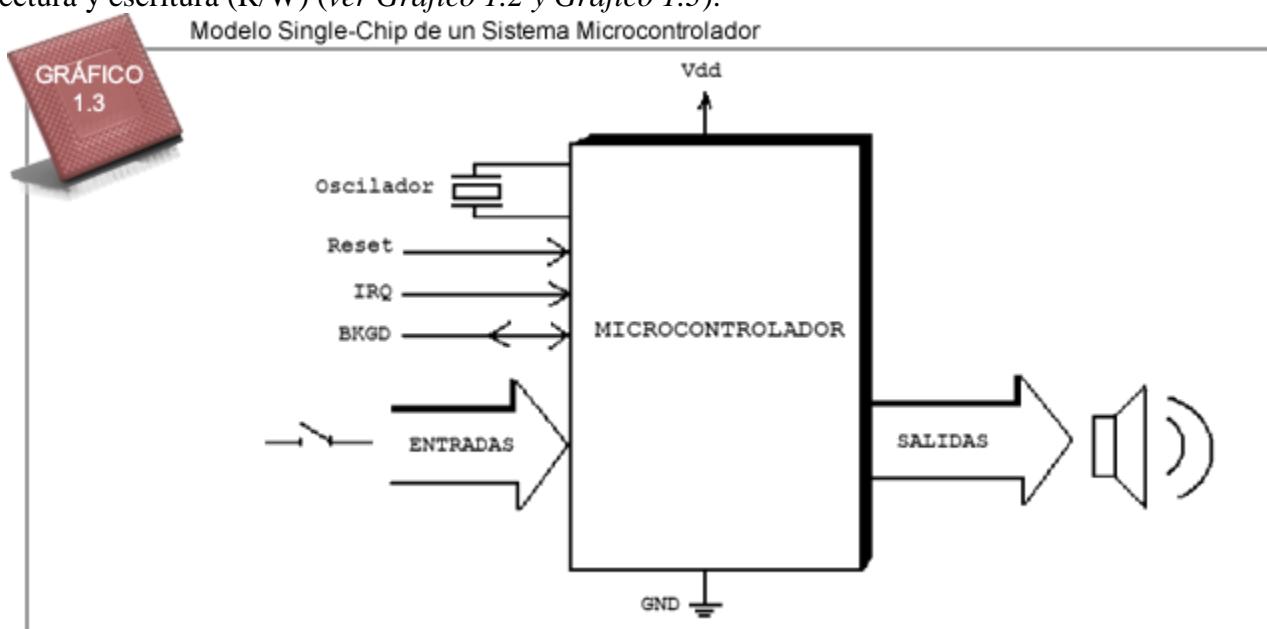
Vdd: Alimentación de voltaje.

Vss: Referencia negativa de voltaje.

CS1: Chip Select de puerto de Entrada.

CS2: Chip Select de puerto de Salida.

El sistema embebido *single-chip* contiene en una sola pastilla de silicio, todo el esquema de un sistema expandido convencional con su procesador en el centro, sistema de decodificación, sistema de multiplexaje datos/dirección, memorias de almacenado (en la cual es ejecutado el programa), memoria RAM, además de las señales de sincronización que administran el acceso a las memorias y los ciclos de lectura y escritura (R/W) (*ver Gráfico 1.2 y Gráfico 1.3*).



1 Imagen cortesía de Freescale™.

1.2.1 ¿Qué es un compilador? ¿Qué es un interpretador?

El programa editado es llevado a la máquina que lo ejecutará finalmente, en este caso un microcontrolador comúnmente llamado “*target*”, de dos formas posibles:

- El programa es convertido a código de máquina al 100% y luego se ejecuta.
- El programa es enviado al “*target*” tal cual fue editado, y éste toma línea por línea y lo va convirtiendo a código de máquina a medida que va pasando por él.

En el primer caso se dice que el código es **compilado**, y en el segundo caso, que el código es **interpretado**.

Un **compilador** es un programa de software que transforma uno o varios archivos de código fuente y genera un archivo en código de máquina llamado ejecutable; este nuevo archivo es enviado al “*target*” para que lo ejecute.

Un **interpretador** es un software que es instalado en el “*target*”, el cual está preparado para recibir un archivo fuente editado; una vez el “*target*” recibe la orden de ejecutar inicia el proceso de cambiar línea por línea de programa a su respectivo código de máquina y ejecutar este código (*ver Gráfico 1.4*).

Ambos esquemas tienen ventajas. Para el caso de un compilador el tiempo requerido para convertir el código que ejecutará la máquina se utiliza completamente antes de su ejecución final, y en el caso de compiladores cruzados, este código lo convierte una máquina con mayores recursos que el “*target*”, así el código queda listo para ejecutarse ahorrando tiempo a la máquina final, la cual no se tiene que enterar de los pormenores del código. Sin embargo, para hacer la conversión a código de máquina, se deberá conocer con anterioridad la máquina que va a ejecutar el código, y este código quedará limitado a ejecutarse en dicha máquina.

En el caso del interpretador, se tiene el código fuente hasta el último momento, cuando es ejecutado y el “*target*” se encarga de la conversión, de modo que el código no queda limitado a ejecutarse en determinada máquina, sino que quedará libre para que cualquier máquina lo tome, lo convierta y lo ejecute. La gran ventaja en este caso es su amplia portabilidad, lo que independiza el código de la máquina que lo ejecuta; sin embargo, su gran desventaja radica en que la máquina tendrá que invertir parte de su tiempo y recursos en tomar el código y convertirlo a su respectivo código de máquina para luego hacer su ejecución.

Una vez que el programa está compilado las líneas del código fuente dejan de tener sentido en la ejecución del programa. Cuando se usa un intérprete, el código fuente debe estar presente cada vez que se quiere ejecutar el programa.

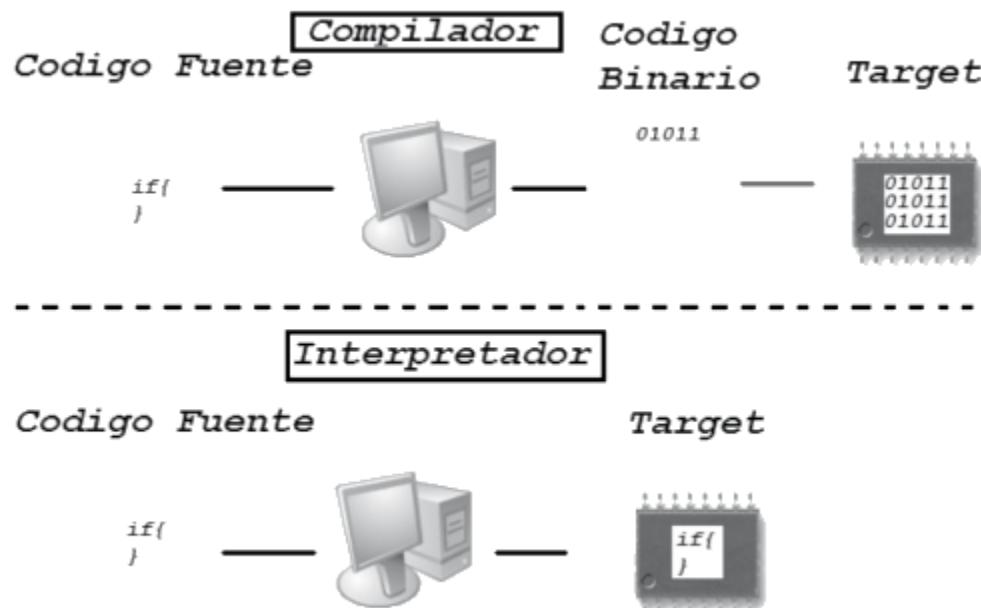


Un archivo compilado se ejecuta directamente en el microcontrolador porque está convertido a lenguaje máquina; en cambio, un archivo interpretado requiere un software incluido en el microcontrolador para su traducción.





Ejecución de código en Compilador y en Interpretador.



1.2.2 Estructura y pasadas de un compilador



Para sistemas embebidos es preferible usar compiladores a interpretadores, debido a las limitaciones en tiempo de ejecución del procesador.

El compilador, como programa de software, seguirá varios pasos para realizar la conversión del código texto original .C y .H a su código ejecutable, en este caso .S19 o .HEX, que será el archivo que se enviará al sistema final *target*".

Los archivos con extensión .C y .H que corresponden al proyecto, son inicialmente pasados por un subprograma llamado el **pre-procesador**, el cual se encarga de crear archivos intermedios con la solución de todos los macros (conversión de texto en números o equivalencias), solucionar la compilación condicional (en caso de existir), e incluir el código que finalmente se compilará (*ver Gráfico 1.5*).

Este archivo temporal se pasa a otro subprograma encargado del **Análisis Semántico**, donde se analiza línea a línea el código, verificando que todos los paréntesis y/o corchetes abiertos sean cerrados, que las palabras reservadas estén correctamente escritas o, lo que es lo mismo, que todas las palabras estén en el léxico del C.



En una compilación el pre-procesador

Si este subprograma detecta algún error, suspenderá su entrega al subprograma siguiente y dará como salida un archivo de errores (típicamente de extensión .ERR), el cual es usado para mostrar en el ambiente del editor uno a uno los errores que el compilador encuentra en este paso.

Si por el contrario, no se tienen errores sintácticos, se envía el nuevo código intermedio al **Generador de Código**, el cual tendrá como salida un código en ensamblador (extensión .ASM), equivalente línea a línea al código de entrada entregado.

Este nuevo archivo en lenguaje de máquina .ASM, es pasado por otro subprograma denominado el **Optimizador**, este tomará el archivo y realizará un análisis global al código y dependiendo de las optimizaciones señaladas por el programador, obviará algunas líneas de código, modificando así el código de máquina y generando un nuevo código más corto, más rápido y con la misma funcionalidad del archivo original. Este Optimizador generará a su vez, además del archivo .ASM optimizado, un archivo paralelo de extensión .LST que contiene la equivalencia entre el código ASM y el Código original.

El nuevo archivo ensamblador .ASM pasará por el programa **Ensamblador**, el cual generará el código objeto (extensión .OBJ o .O), que es un código en lenguaje de máquina codificado de los archivos del proyecto.

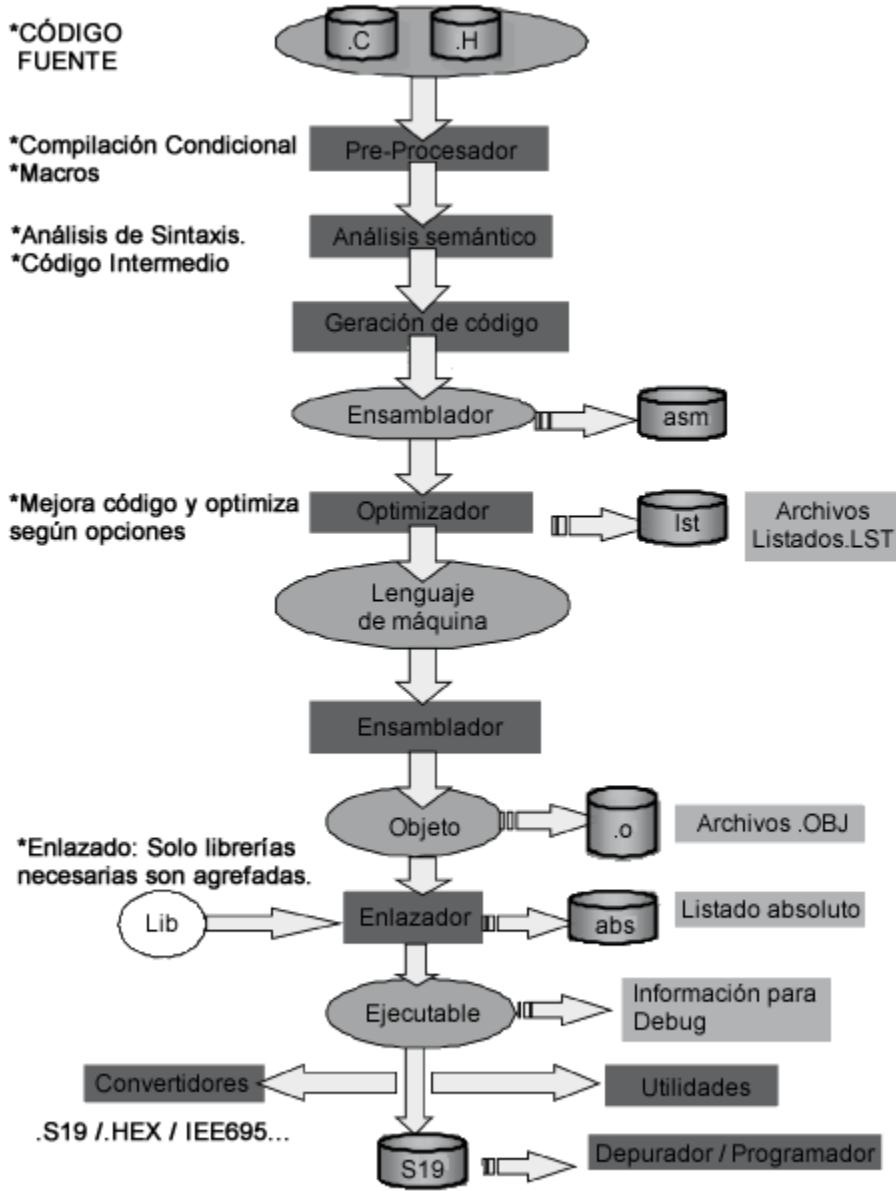
Este archivo es luego enlazado por medio del *linker* o **Enlazador**, el que agregará el contenido de las liberarías de C que se invocaron en el proyecto, y de esta forma, generará un código ejecutable posicionado en las direcciones específicas que indica el proyecto, creando el archivo final .S19, .HEX (o ejecutable) y otros archivos adicionales de ayuda para la depuración, sea en tiempo de simulación o en tiempo real.

crea archivos intermedios, los cuales pasan al subprograma de Análisis Semántico y de ahí al Generador de Código; luego, un Optimizador lo analiza y elimina las líneas innecesarias para generar un archivo .ASM en lenguaje máquina.

Todo archivo .ASM optimizado debe pasar por el programa Ensamblador, el cual generará el código objeto .OBJ, que será vinculado a las direcciones específicas indicadas por el proyecto a través del programa Enlazador, originando un archivo final .HEX o ejecutable.

GRÁFICO
1.5

Estructura y pasadas de un compilador estándar.



1.2.3 Tiempo de compilación vs tiempo de ejecución

El **tiempo de compilación** (tc en *Gráfico 1.6*), es el tiempo que tarda el compilador instalado en el PC, para realizar la conversión de todas las líneas de código y pasarlo a lenguaje de máquina; esta conversión incluye la solución de todos los símbolos y operaciones que son posibles resolver. Una vez ejecutado, este tiempo no requerirá ser utilizado por el procesador final “*target*”.

Para el caso de un interpretador, todo el código deberá ser

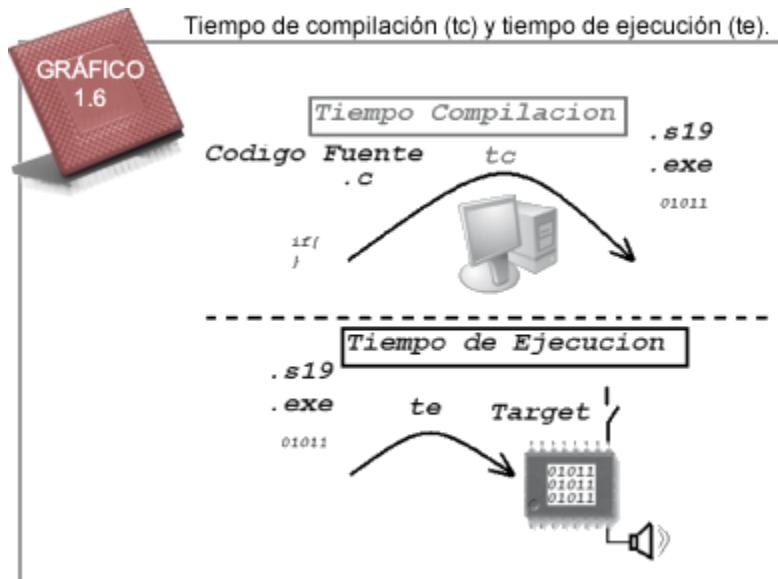


Cuando se compila un archivo el software debe traducir las líneas del

utilizado en tiempo de ejecución, y el tiempo de compilación será nulo; como sabemos, el código tal como fue digitado es pasado al procesador final para su completa ejecución.

El **tiempo de ejecución** *te* en *Gráfico 1.6* es el tiempo que toma el “*target*” en realizar una operación o resolver alguna expresión, previa evaluación de cada una de las variables involucradas en el proceso.

código a lenguaje máquina, lo que tarda dicha operación se conoce como tiempo de *compilación*.



Es importante identificar en el programa cuales expresiones serán ejecutadas en tiempo de compilación y cuales lo serán en tiempo de ejecución o de procesamiento del “*target*”. Las primeras ayudan a dar claridad al programa sin sacrificar el procesamiento, todos los macros realizados mediante la directiva #define no adicionaran tiempo de ejecución, igual pueden hacerse declaraciones que nunca se usen y el compilador no generara código para ellas.

```
#define XTAL_EXTERNO 4000000 // [Hz] frecuencia del oscilador externo.
```

```
#define FREC_BUS XTAL_EXTERNO/2 // [Hz] cálculo de la frecuencia del bus interno del procesador.
```

```
#define PRESC_SCI 1 // Divisor de Prescaler
```

```
#define BAUD_RATE_DIV 4 // Divisor de Baud Rate
```

```
#define BAUD_RATE FREC_BUS/( 64*PRESC_SCI *BAUD_RATE_DIV)
```

La anterior división en la definición de **FREC_BUS** es resuelta en tiempo de compilación, debido a que es posible definir el valor de la constante y no depende de la evaluación de una variable.

Con esto, en cualquier expresión que se haga referencia a **FREC_BUS**, no se remitirá a la operación para obtener su valor, sino a su valor ya calculado por el compilador, pasándole al “target” la constante ya evaluada.

1.3 ¿DE DÓNDE VIENE EL ANSI C?

El lenguaje C fue diseñado en 1972 por los ingenieros del Laboratorio Bell, entre ellos, Denis **MacAlistair Ritchie**. La razón principal fue reescribir el sistema operativo UNIX en una forma más portable, debido a que el UNIX fue escrito originalmente en lenguaje ensamblador y debía ser completamente reescrito para cada nueva máquina. El lenguaje C fue basado en previos lenguajes llamado B y BCPL y por esta razón el llamarlo C denota su evolución.

EL lenguaje fue presentado por la publicación “*The C Programming Language*” en 1978, cuyo “Appendix A” ha sido usado por muchos años como estándar. Dado el crecimiento y popularidad del UNIX y la gran aceptación del lenguaje C, muchas compañías empezaron a proveer compiladores de C de forma alternativa al UNIX para un gran número de procesadores, microprocesadores e incluso, para microcontroladores.

El éxito del C en pequeños procesadores se debe al hecho que no es un lenguaje de tan alto nivel como lo es el PASCAL o el ADA, sino un macro ensamblador altamente portable, lo que permite tener la misma flexibilidad que el lenguaje ensamblador ofreciendo casi el mismo nivel de eficiencia.

El C fue finalmente normalizado entre 1983 y 1990 y es ahora un estandard ANSI/ISO (Por sus siglas en inglés: *American National Standards Institute* www.ansi.org¹/ *International Organization for Standardization* www.iso.org). Este estándar implementa varias extensiones adicionadas al lenguaje original en la década previa, y algunas características extras, permitiendo mayor robustez.

El estándar ANSI/ISO distingue dos ambientes diferentes:

El ambiente **hosted**, que describe a los compiladores nativos (*native compilers*), en los cuales el procesador que ejecuta el código usa un sistema operativo específico con manejo de archivos y diversos ambientes de ejecución.

El ambiente **freestanding**, que describe a los compiladores cruzados (*cross compilers*), en los cuales el procesador que ejecuta es generalmente un microprocesador embebido o un microcontrolador sin los servicios de algún sistema operativo.

En el sistema nativo (**hosted**), la misma máquina es usada para crear, compilar y ejecutar el programa en C, mientras que en el sistema embebido, la aplicación no tiene archivos, sistema operativo, ni editor, en su lugar otra máquina ha de ser usada para crear y compilar la aplicación. El resultado se envía a la máquina



El lenguaje C fue desarrollado en 1972 por un grupo de ingenieros de Laboratorios Bell, liderados por Denis MacAlistair Ritchie.

El estándar ANSI llevó luego a la normalización ISO/IEC 9899:1990², la cual garantiza que todas las aplicaciones desarrolladas bajo esta norma funcionarán correctamente en cualquier plataforma con una implementación de C compatible.



definitiva “*target*” para que lo ejecute.

La máquina usada para compilar la aplicación es llamada el sistema *host*, y la máquina que ejecuta la aplicación es llamada el sistema “*target*”.

2 Ver información de la especificación en <http://www.iso.org>

1.3.1 Forma general de un proyecto en C

Un proyecto en C por lo general está dividido en varios archivos, cada uno de los cuales contiene una parte del texto completo de la aplicación. Algunas partes están previamente escritas y son usadas como librerías, otras pueden estar escritas en lenguaje ensamblador donde el C no sea lo suficientemente óptimo, o no permita el acceso directo a los recursos del procesador.

Archivos en C



Un proyecto desarrollado en C comprende las líneas de código, los comentarios del programador, los identificadores, las palabras reservadas, las constantes y los operadores y signos de puntuación.



Cada uno de los archivos del proyecto deberá ser compilado, el compilador transforma el código texto en un archivo llamado objeto relocalizable (aun sin dirección fija en la memoria del microcontrolador). Una vez los archivos son compilados, el programa “*linker*” o enlazador, es usado para agrupar en un todo los archivos resultantes, incluyendo las librerías, para así producir el archivo ejecutable, el cual contiene un formato específico para poder ser transferido a la máquina que finalmente lo ejecutará.

Línea de Código

Cada texto en C contiene un conjunto de líneas y cada línea contiene a su vez caracteres y es finalizada con una nueva línea (un *Line Feed* y un *Return*). El compilador de C permite que varias líneas sean concatenadas en una sola línea, con una longitud que no exceda los 511 caracteres para cumplir estrictamente con el estándar ANSI C.

Comentarios

Los comentarios son parte del texto y aunque no tienen ningún significado para el compilador son importantes porque brindan la posibilidad de entender el código, lo que a su vez facilita su modificación.

Un comentario inicia con la secuencia /* y finaliza con la secuencia */. Un comentario puede cubrir varias líneas; sin embargo, los comentarios anidados no son permitidos. Como una extensión del estándar ANSI C la mayoría de los compiladores aceptan los comentarios en estilo C++, los cuales inician con la secuencia // y finalizan con una nueva línea de código.

```
A = b; // este es un comentario
```

```
C = d; /* este es otro tipo de comentario
```

Que cubre varias líneas de C*/

Identificadores

Un identificador es usado para dar nombre a un objeto. Este empieza por una letra o el carácter “*underscore*” _, seguido por una letra o un dígito. Los identificadores son sensibles a mayúscula y minúscula, de tal manera que **vBle1** es diferente de **vBLE1** e identifican objetos diferentes. Un identificador puede contener hasta 255 caracteres aunque en la práctica no se recomiendan identificadores de más de 20 caracteres.

Palabras reservadas

Una palabra reservada es un identificador usado por el lenguaje de programación para describir algo especial y propio de dicho lenguaje. Es usado en declaraciones para describir el tipo básico de un objeto, o dentro de una función para describir la ejecución de una secuencia.

Una palabra reservada no puede ser usada como nombre de objeto. Todas las palabras reservadas se escriben en minúsculas, de tal manera que la versión en mayúscula esta disponible como nombre de objetos, aunque no es una buena práctica de programación declarar estas palabras reservadas como nombres.

Las palabras reservadas son:

```
auto double int struct  
break else long switch  
case enum register typedef  
char extern return union  
const float short unsigned  
continue for signed void  
default goto sizeof volatile  
do if static while
```

Constantes

Una constante es usada para describir un valor numérico o una cadena de caracteres. Las constantes numéricas pueden ser expresadas como constantes reales, constantes enteras o de carácter. Las constantes enteras pueden ser expresadas en base decimal, binaria, octal o hexadecimal.

Operadores y signos de puntuación



En C un objeto es un identificador de nombre a un objeto para el programador y puede ser bautizado con cualquier combinación de caracteres siempre que no sean las palabras reservadas.



Un operador es usado para describir una operación aplicada a uno o varios objetos. Los operadores son ampliamente usados en muchas expresiones y en diversos tipos de declaraciones. El compilador tratará de solucionar las operaciones que le sean posibles en tiempo de compilación, previniendo así que sea el “target” quien lo deba hacer.

Los operadores consisten en una secuencia de caracteres no alfanuméricos (uno en su mayoría, a veces dos). Un signo de puntuación se usa para separar o terminar una lista de elementos.

Operadores y signos de puntuación en C.

TABLA 1.1

Operadores Aritméticos	+ - * / % = += -= *= /= ++ --
Operadores Booleanos	& ~ ^ == && = &= != ? ! ^=
Operadores de Comparación	> < >= <=
Operadores de Desplazamiento	>> << <<= >>=
Manejo de apuntadores	-> & *
Paréntesis, Corchetes	[] { } ()
Puntuación	; : ,
Compilación y Declaración	# ##

Nótese que algunos de ellos son usados como operadores en algunos casos, y en otros como signo de puntuación. Algunos de ellos son utilizados por pares, tal es el caso de (), [], { }.

1.4 PERIFÉRICOS MÁS COMUNES EN SISTEMAS EMBEBIDOS

La gran mayoría de los microcontroladores del mercado, y dependiendo del derivativo específico y de la aplicación, incluyen varios periféricos internos, los cuales facilitan la integración de un sistema y hacen que la aplicación final sea compacta, pequeña, optimizada para bajo consumo de energía, de pocos elementos externos y por consiguiente, de bajo costo.

A continuación se hará una breve descripción de los periféricos internos asociados a los procesadores de 8 bits, el objetivo es conocer sus características generales y la información básica para hacer uso de ellos en una aplicación real.

1.4.1 Puertos de entrada/salida y Función KBI

Este periférico es la interfaz entre el bus de datos interno y los pines asignados a funciones de entrada y salida del microcontrolador. Al momento del *reset* los pines bidireccionales inician configurados como pines de entrada, para evitar que su estado de salida active los periféricos externos al microcontrolador



El puerto
de entrada/

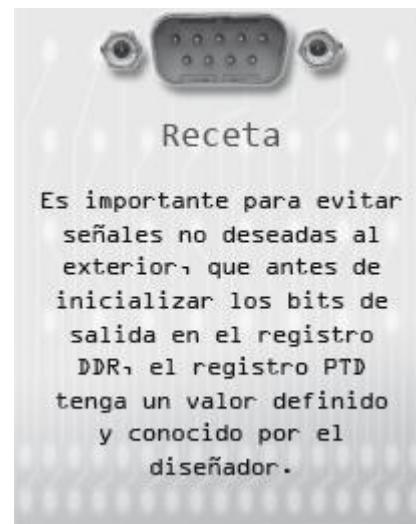
o genere choque de datos con los periféricos externos.

Es tarea del diseñador configurar el sentido, ya sea de entrada o salida de cada uno de los pines dependiendo del hardware asociado a cada uno de ellos, lo mismo el valor inicial que tomará el pin una vez se configure su sentido.

Para el control y manejo de cada puerto existen principalmente dos registros:

1. En el registro de DATO EN EL PUERTO (**PTD**) está el dato (bit a bit) del estado de cada uno de los pines en el exterior: si el pin es de salida, el bit correspondiente tendrá el estado que el MCU escribe al pin en el exterior; si el pin es de entrada, en el bit del puerto estará el estado que lee del exterior.
2. El registro de DIRECCION DEL PUERTO (**DDR**) define bit a bit el sentido, ya sea entrada o salida de cada uno de los pines. Dependiendo de la tecnología y marca específica, algunos definen que un CERO en el bit define el pin como salida, mientras que en otras (como es el caso de Freescale™), define que un UNO en el bit correspondiente define que el pin será de salida (*ver Gráfico 1.7*).

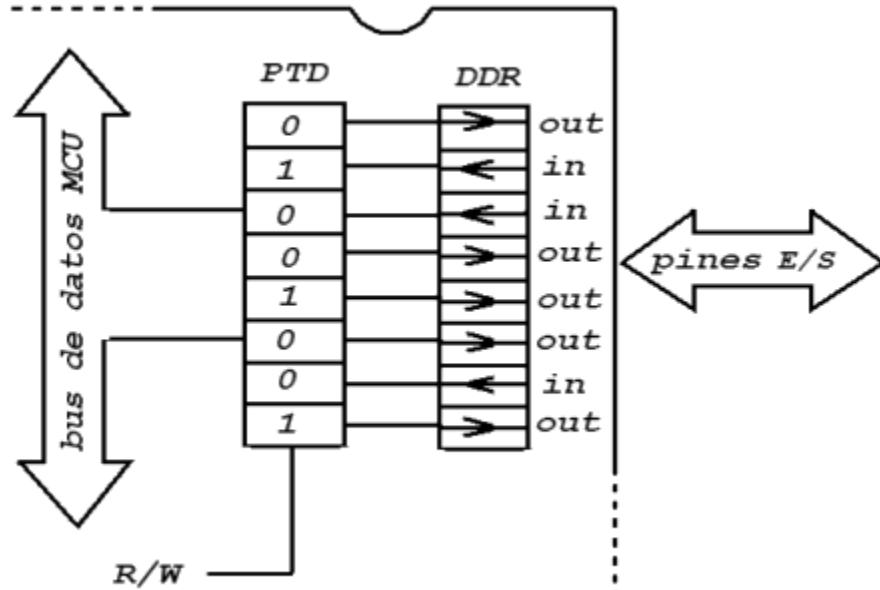
salida es uno de los principales periféricos internos de todo microcontrolador. Permite intercambiar información entre el bus de datos interno del microcontrolador y otros dispositivos.



Es importante para evitar señales no deseadas al exterior, que antes de inicializar los bits de salida en el registro DDR, el registro PTD tenga un valor definido y conocido por el diseñador.



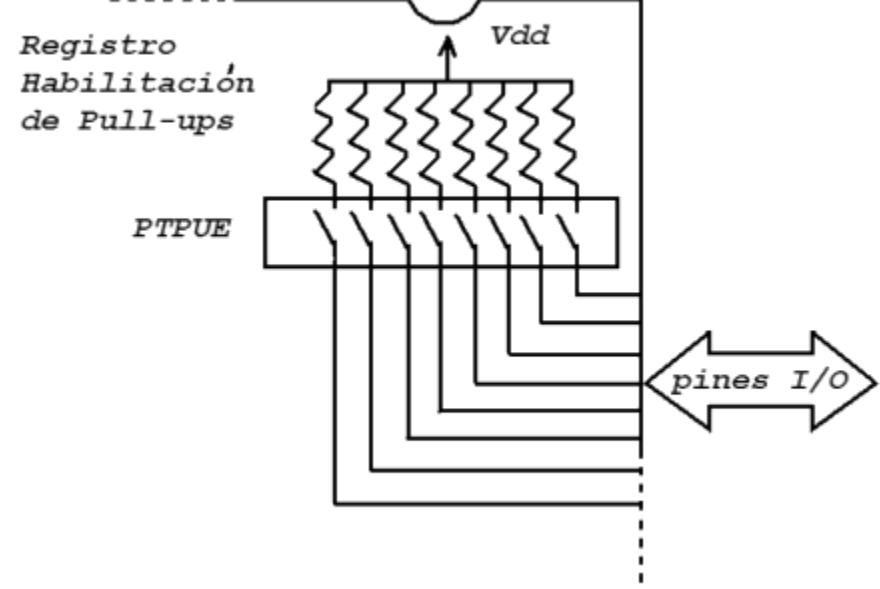
Estructura interna puertos de entrada/salida.



Existen algunos puertos que adicionalmente contienen un registro adicional para habilitar una resistencia de pull-up en los pinos de entrada/salida (**PTPUE**), estos permiten disminuir el hardware externo al MCU (ver Gráfico 1.8).



Registro de Pull-Up.



1.4.2 Conversor analógico a digital (ADC)

Consiste en un módulo que permite al microcontrolador recibir una señal analógica del mundo exterior, cuyos límites de voltaje oscilan entre 0 voltios hasta el nivel de alimentación, y convertirla en un valor digital equivalente.

El resultado estará disponible en un registro interno del microcontrolador, el cual puede leerse, procesarse o almacenarse para un análisis posterior, permitiendo al software tomar decisiones de control dependiendo del comportamiento de la señal analógica.

De acuerdo al derivativo varían según su resolución, pueden ser de 8 bits, 10 bits (típico) o 12 bits.

La técnica comúnmente usada para la conversión es la denominada “aproximaciones sucesivas SAR”, y puede configurarse para que realice conversiones sucesivas sobre la señal externa o conversión única, dependiendo de los requerimientos de muestreo de la señal.

Este sistema interno de aproximación, es un circuito secuencial, por lo que requiere una señal de sincronización (reloj) que se proporciona de forma interna. Su frecuencia es seleccionable para permitir flexibilidad entre rapidez de conversión y consumo de energía.

Una vez se han generado los ciclos necesarios para terminar la conversión de una muestra analógica, la CPU es notificada mediante el cambio de un bit en el registro de estado del ADC (en 1 indicará que la conversión está finalizada), y opcionalmente puede generar una interrupción de fin de conversión, que le permite al sistema central realizar labores alternas o permanecer en un modo de bajo consumo hasta que se realice la conversión.

Típicamente un microcontrolador contiene un solo sistema de ADC, lo que NO obliga a que solo soporte una señal analógica externa. La entrada del ADC es multiplexada con varios de los pines externos del microcontrolador denominados canales de ADC, con esta técnica pueden conectarse varias señales al chip, las cuales comparten la arquitectura interna del ADC y registros (*ver Gráfico 1.9*).

La relación entre el valor externo y la conversión es completamente lineal; la señal externa, cuyos valores estarán entre 0 voltios y el voltaje de alimentación, darán como resultado un valor entre 0x00 y 0xFF para la resolución de 8 bits, o entre 0x0000 y 0x03FF para el caso de 10 bits.

Los registros que permiten hacer uso del ADC en los microcontroladores de Freescale™ son los siguientes:

ADSCR → (1 byte). Registro de estado y configuración de parámetros del reloj y del canal a convertir.

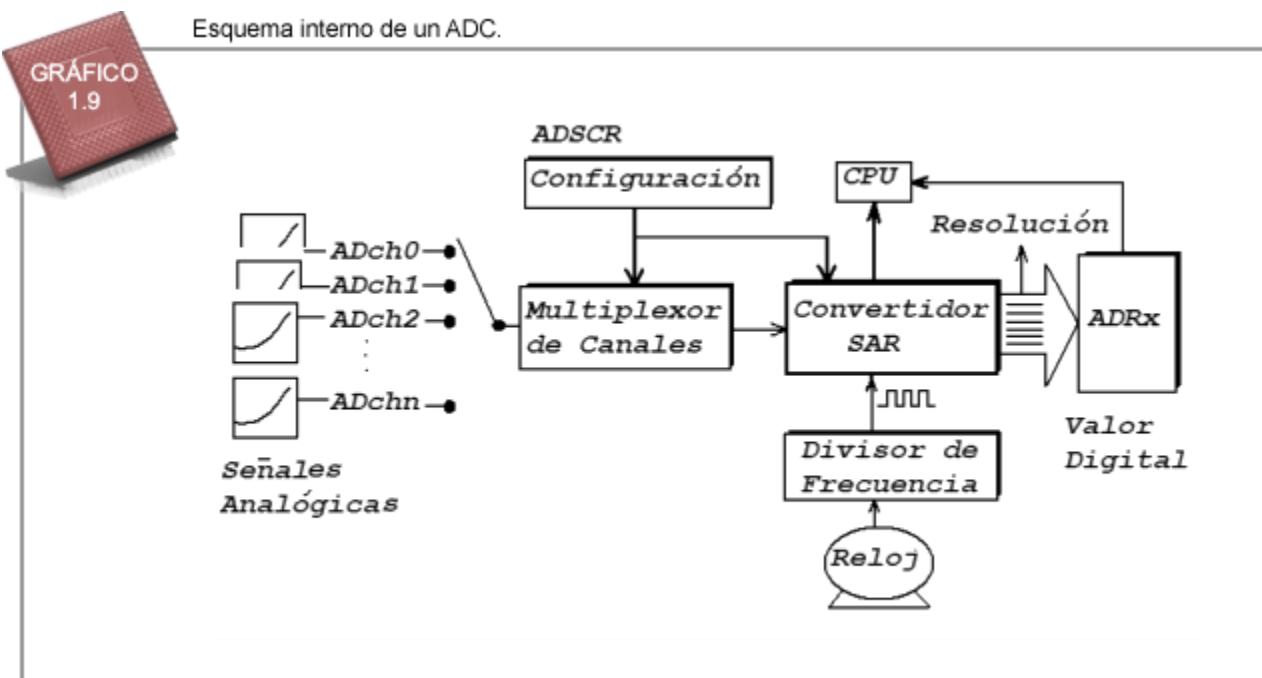
ADICLK → (1 byte). Registro de selección de frecuencia del reloj del ADC.



El conversor analógico a digital (ADC) permite que el microcontrolador reciba una señal análoga externa (una temperatura) y las convierta en datos digitales (un valor interno que representa su nivel).



ADR_x → (2 bytes). Registro donde es almacenado el resultado de la conversión del canal. x (x entre 0 y el número máximo de canales-1 del ADC).



1.4.3 Computador operando apropiadamente (COP)

También llamado perro guardián (*watchdog*). Es un módulo de protección OPCIONAL contra errores de bloqueo del software, ya sea por ruido o por alguna situación de diseño del *firmware* que lleve al procesador a un ciclo infinito

Consiste en un registro contador que al rebosarse produce un *reset* en la máquina recuperándola del estado desconocido, e indicando en un bit de un registro interno ésta como la causa del *reset*.

El software del usuario deberá hacer el borrado del contador antes que rebose, para prevenir el *reset* de la máquina por este concepto.

La rapidez con la que el contador realiza su rebosamiento es seleccionable por el programador, dependiendo de los tiempos máximos de ejecución. En este caso es recomendable usar la frecuencia más lenta de conteo, para permitir al sistema mayor tiempo de recuperación por vías normales (ver Gráfico 1.10).

El uso típico de este módulo es la protección de la aplicación, aunque en algunas aplicaciones de muy bajo consumo de energía se utiliza como contador de tiempo, ya que si el contador de protección no es borrado, se produce un *reset* de forma periódica.

Al salir de *reset* el microcontrolador Freescale™ arranca con el COP HABILITADO, con lo que el software del usuario deberá



El módulo COP evita que el procesador llegue a un ciclo infinito que deshabilite su funcionalidad. En síntesis, es un registro contador que produce un *reset* en la máquina

recordar el borrado del contador de rebosamiento periódicamente.

El programador puede (si así lo desea), deshabilitar el sistema simplemente cambiando el estado de un bit en el registro de configuración del COP y con esto no será necesario hacer el borrado del contador de rebosamiento, y se podrá ahorrar un poco mas de energía y evitar *resets* indeseados.

antes de llegar a
dicha situación.



Es recomendable por orden de programación, que el borrado del contador se realice en una sola línea del código y de preferencia hacerlo en el módulo principal de aplicación en la función **main(){};** solo en algunos casos muy especiales o donde se pronostique que el software pueda demorarse, se recomienda hacerlo en rutinas particulares, pero nunca en rutinas de interrupción, las cuales están condicionadas a suceder bajo circunstancias muy específicas.

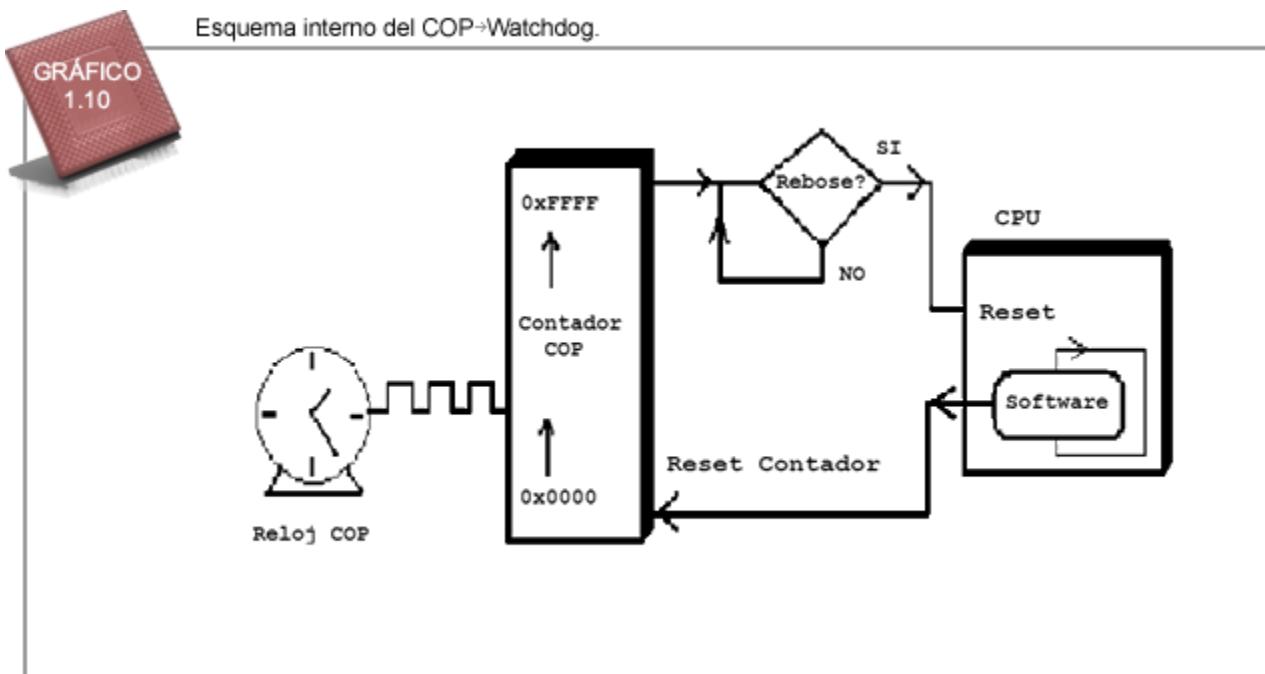
Como se mencionó, este módulo es de protección, por lo que en la etapa de diseño se recomienda deshabilitarlo para evitar el estar pendiente del borrado del contador de rebosamiento, facilitando la programación, dando libertad al programador y facilitando la depuración. Una vez el software tiene una versión final, se habilita el COP, proporcionando una protección adicional al sistema completo.

Los registros típicos que permiten la configuración del COP son:

COPCTL → Registro de borrado del contador de rebosamiento.

CONFIG1 → Registro que permite habilitar o deshabilitar el COP y el período de rebosamiento del contador.

SRSR → Indica en el bit el COP como causa de reset.



1.4.4 Detector de bajo nivel de voltaje (LVI)

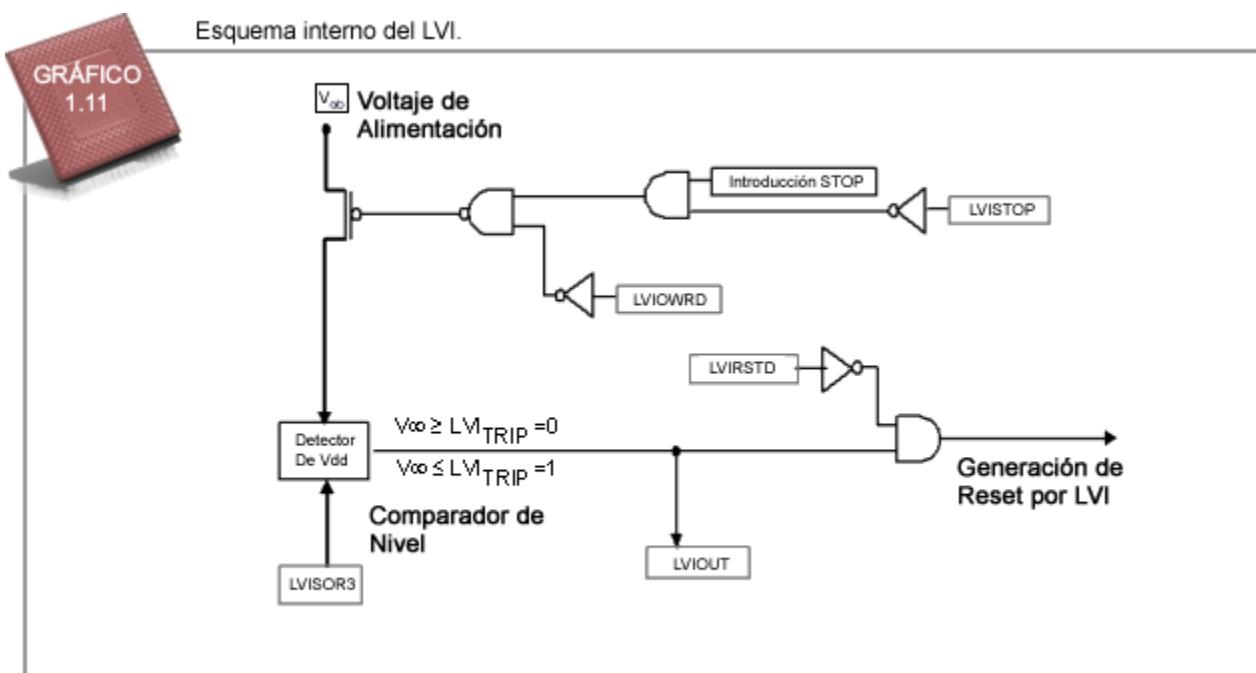
Este módulo proporciona al sistema microcontrolador una protección, mediante la cual se evita que la CPU y los periféricos operen por debajo de los niveles de voltaje de alimentación indicados por el fabricante.

El módulo LVI previene que el microcontrolador opere por debajo del voltaje mínimo indicado por el fabricante.

Debido a que es un sistema de protección, este es opcionalmente encendido y puede ser deshabilitado o habilitado en funcionamiento normal mediante el bit LVIPWR; cuando se requiere ahorrar energía resulta de beneficio deshabilitar el módulo o solicitarle que una vez la CPU ejecute la instrucción STOP, sea apagado de forma automática, control realizado mediante el bit LVISTOP.

El LVI una vez habilitado, comparará el nivel del voltaje de alimentación, si este nivel está por debajo del nivel permitido y el bit de LVIRSTD está habilitado, generará un reset en la CPU, y lo mantendrá en este estado hasta que el voltaje de alimentación regrese de nuevo a los niveles considerados como apropiados.

En la mayoría de los microcontroladores Freescale™, el nivel de voltaje a detectar se puede seleccionar entre 2 valores: 3 ó 5 voltios mediante el bit LVI5OR3, dependiendo del voltaje de alimentación al cual este trabajando el microcontrolador (*ver Gráfico 1.11*).



1.4.5 Temporizador (TIMER)

Es un módulo que facilita manejar el tiempo. Puede configurarse para generar interrupciones periódicas a una frecuencia seleccionada, lo cual permite que el procesador pueda disponer de un reloj de tiempo real a bordo.

La frecuencia a la que sucede cada interrupción es seleccionable dependiendo de la resolución que requiera el reloj de tiempo real. Bajo ciertas configuraciones permite continuar su labor de interrupción, aun si la CPU entra a algún modo de bajo consumo: STOP o WAIT.

El temporizador, comúnmente llamado “*timer*”, tiene asociadas varias funciones adicionales para interactuar con el hardware externo, permitiendo realizar medidas de señales conectadas a pines del microcontrolador o generación de señales temporizadas.

La función del “*timer*” está basada en un contador maestro de 16 bits que esta permanentemente incrementándose, si llega a su valor máximo pasa a cero y así repite el proceso nuevamente.

Este sobreflujo también genera una interrupción a la CPU para indicar este evento y no perder la noción del tiempo y permitir así contabilizar tiempos mayores al rango de 16 bits. De forma opcional puede generar una acción sobre un pin externo, cambiando su estado e indicando que el temporizador ha sido reiniciado, esta función resulta útil en aplicaciones de PWM³ que se verán a continuación.

Funciones asociadas al temporizador

Entrada de captura: “Input Capture”

Esta función es usada para que de forma automática se tenga en un registro de 16 bits, la marca de tiempo exacta en la que sucedió un evento en un pin externo, ya sea cambiar de nivel lógico 1 a 0 (flanco de bajada) o pasar de nivel lógico 0 a 1 (flanco de subida).

La función de entrada de captura es muy útil para medir de forma exacta la frecuencia externa de una señal digital, ya que al medir 2 flancos seguidos obtendremos el período de la señal, y con una operación matemática su frecuencia, o igual para medir el ciclo útil (*Duty Cycle*) de una señal externa, conociendo el período y la duración entre un flanco de subida y uno de bajada (*ver Gráfico 1.12*).

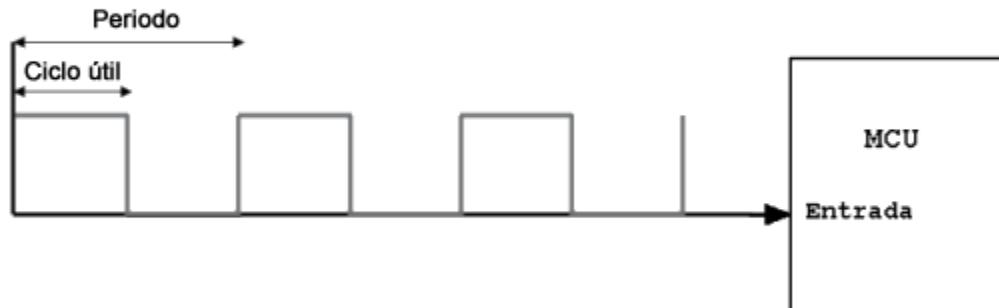


El temporizador regula el manejo del tiempo en el microcontrolador. Lo hace por medio de un contador maestro de 16 bits que se incrementa continuamente, y que al llegar a su máximo valor pasa a cero y repite el proceso indefinidamente.



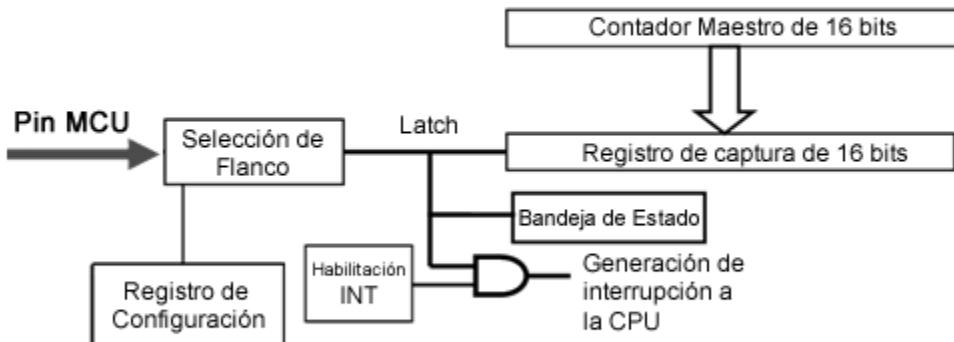
**GRÁFICO
1.12**

Señal digital externa aplicada al microcontrolador para medir su frecuencia.



**GRÁFICO
1.13**

Esquema de la función de entrada de captura.



El registro de captura de 16 bits contiene la marca de tiempo exacta en la que estaba el Contador maestro cuando se presentó el flanco seleccionado. Luego la CPU puede acceder a este valor y guardarlo en su memoria RAM, de esta forma el Registro de Captura queda preparado para el siguiente evento ver Gráfico 1.13).

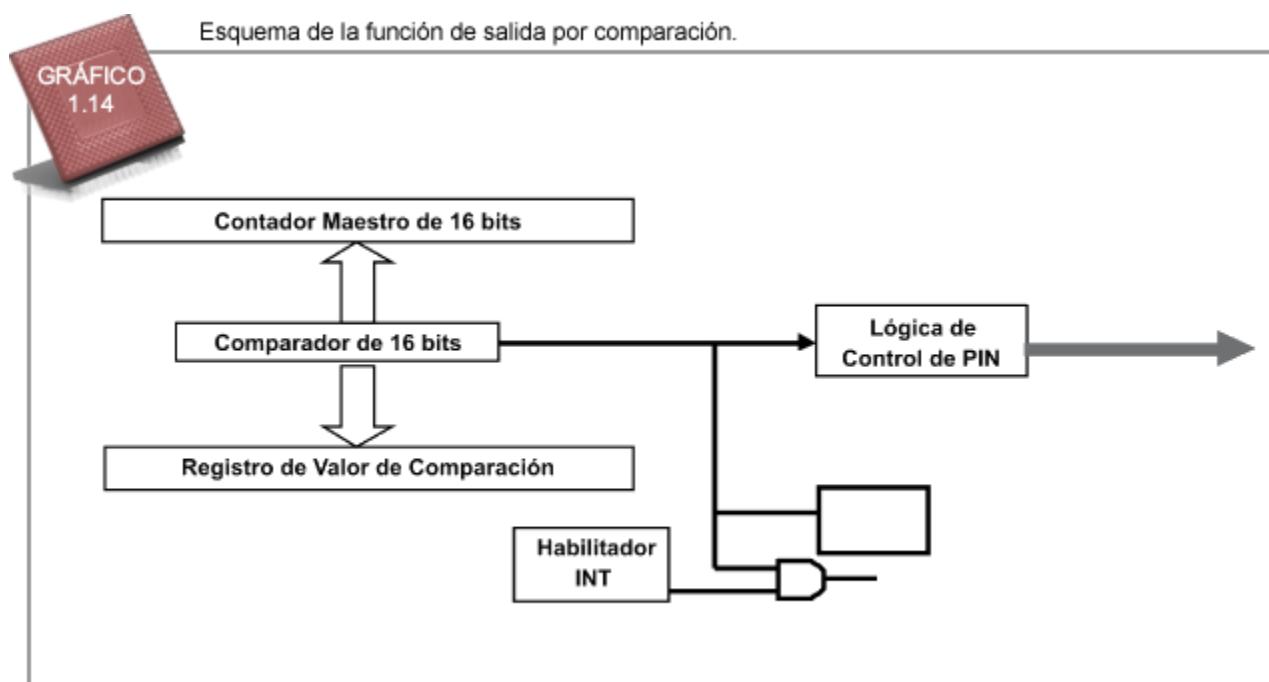
En algunos casos también es utilizada para generar una interrupción a la CPU y avisar sobre un suceso externo.

Salida por comparación: “Output Compare”

Esta función permite actuar sobre un pin externo del microcontrolador en tiempos previamente definidos en un registro, denominado el Registro de Valor de Comparación (*ver Gráfico 1.14*).

Permite la generación de señales digitales que son enviadas al exterior a través de uno de los pines definidos para esta función.

Mediante registros internos se puede decidir la acción que se ejecutará sobre el pin externo una vez se cumpla un tiempo programado, ya sea cambiar el pin de nivel lógico 1 a 0 (flanco de bajada), cambiarlo de 0 a 1 (flanco de subida), o cambiarle de estado actual (*toggle*)⁴.



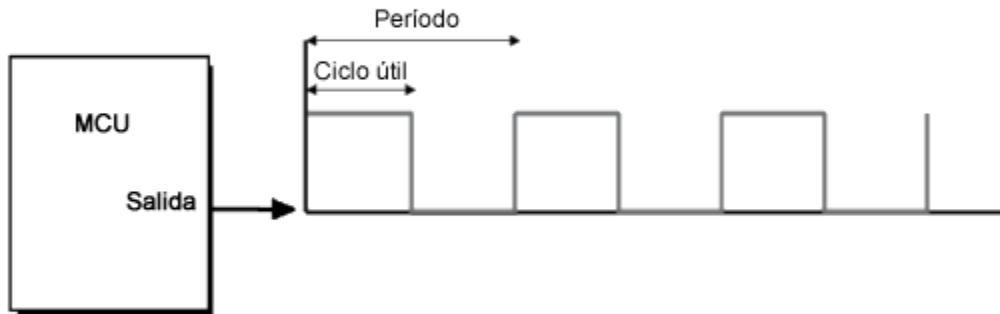
Modulación de ancho de pulso: PWM

El módulo de temporización puede ser configurado de tal forma que genere en uno o varios pines una señal cuadrada con las características de un PWM (Modulación por Ancho de Pulso), la cual es definida como una señal cuadrada que contiene un período fijo y dentro de este un tiempo en estado activo, denominado “*Duty Cycle*” o tiempo de ciclo útil (*ver Gráfico 1.15*).

Esta función amplía las posibilidades de uso de los microcontroladores, dentro de las cuales se incluyen cargadores de batería, conversores digital analógico, controles de intensidad (*dimmers*), control industrial y robótica entre otros.



Señal de PWM generada por el microcontrolador.



Para generar una señal como la del Gráfico 1.15 se considera que el tiempo de ciclo útil se programa en el contador de comparación, configurando que el pin pase a cero (flanco de bajada) una vez se dé la comparación, y para la generación del período se programa como tiempo de desborde del temporizador, con cambio en el valor del pin externo TOVO (*Toggle on Overflow*). De esta forma solo se generarán interrupciones 2 veces por período, una cuando termina el ciclo útil y otra cuando empieza un nuevo período. En esta situación la CPU interviene poco y permite la generación de una onda muy precisa debido a su independencia de la CPU cuando la señal debe realizar su cambio.

Se deben tener consideraciones especiales, como que el ciclo útil nunca ha de ser mayor que el período y también tener precaución de no cambiar de forma brusca el ciclo útil, de lo contrario se puede tener un ciclo inválido que se ajusta en el siguiente período. La mayoría de procesadores con este módulo, ofrecen la opción buffereada, la cual permite que esta situación no se presente entre cambio de período y el cambio de ciclos útil ocurre de forma suave. Si requiere de mayor información sobre esta, puede ser consultada en el manual de cada procesador en particular en la sección PWM.



La modulación de ancho de pulso (PWM), permite obtener una onda con un período muy preciso, la cual puede ser usada como control de intensidad en aplicaciones industriales, pero también en otras de uso cotidiano, como los cargadores de baterías.



- 4 Toggle: cambiar de estado, invertir el estado actual.

1.4.6 Comunicación serial asincrónica (SCI)

El módulo SCI es un subsistema independiente de la CPU que permite, de modo estándar, comunicarse de forma serial con gran cantidad de periféricos externos.

Desde el punto de vista del microcontrolador tiene dos (2) líneas de comunicación, una usada como línea de recepción (Rx) y otra como línea de transmisión (Tx), permitiendo establecer una comunicación *Full Duplex*, es decir, se tiene la capacidad de transmitir y recibir datos de forma simultánea (*ver Gráfico 1.16*).

El SCI usa el estándar NRZ (No Retorno a Cero), que consiste en un bit de arranque (*start*), 8 ó 9 bits de datos y un bit de parada (*stop*). El subsistema contiene un generador programable de velocidades de comunicación (*Baud Rate Generator*), en el cual el programador puede elegir entre 32 velocidades estándar, habilitación independiente del módulo transmisor y receptor, selección de polaridad de los bits y un buffer que evita que en secuencias rápidas se pierda algún dato y un sistema de interrupciones que informa a la CPU sobre los siguientes eventos básicos:

Tx: El buffer del módulo transmisor esta vacío.

Tx: La transmisión de la palabra se finalizó.

Rx: Llegó un dato por la línea de recepción.

Err: Error de ruido en la línea de recepción.

Err: Error de overrun en recepción.

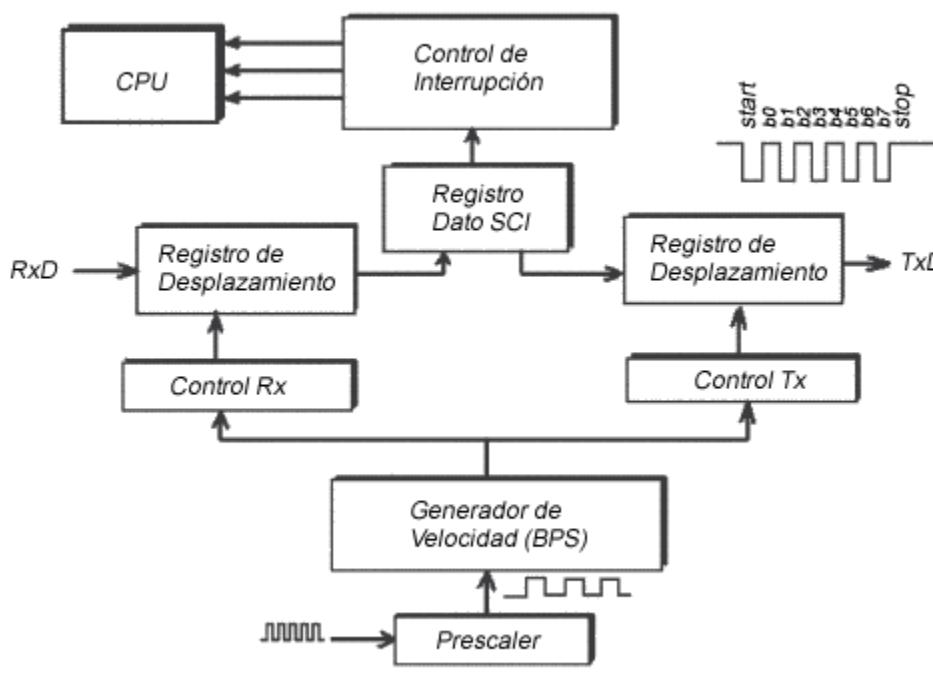
Err: Error de trama (*framing error*).

Err: Error de paridad.

El subsistema SCI permite interfaz con una gran cantidad de módulos periféricos al microcontrolador, donde se incluyen otros microcontroladores, módulos de RF, sistemas de posicionamiento global (GPS), módem celulares, entre otros.

**GRÁFICO
1.16**

Diagrama de bloques del SCI.



1.4.7 Comunicación serial sincrónica (SPI)

Este módulo permite la comunicación serial entre varios dispositivos en un esquema maestro-esclavo. El módulo es útil en aplicaciones donde se requiere optimizar el uso de entradas/salidas, debido a que tiene la bondad de poder interconectar varios dispositivos a través de las mismas líneas de comunicación.

El sistema SPI permite operar bajo control de software en varios modos según la topología seleccionada:

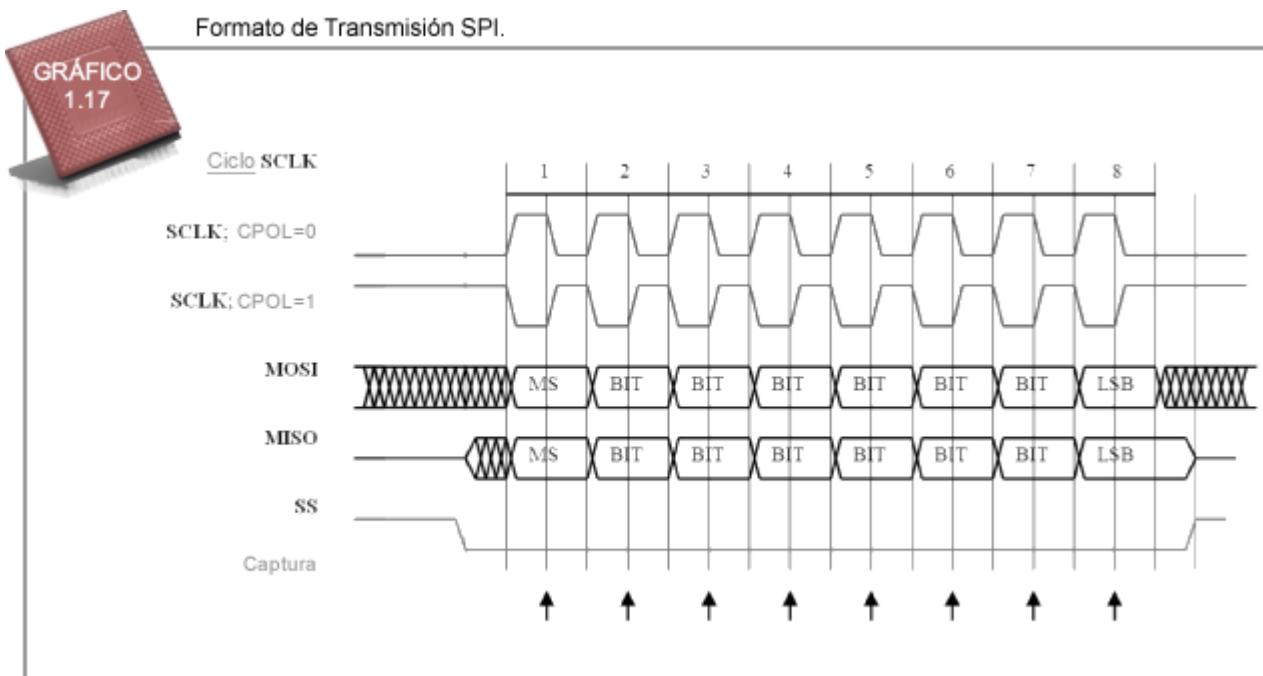
- Un MCU maestro y varios MCUs esclavos.
- Varios MCUs interconectados en un sistema multimaestro.
- Un maestro MCU y uno o varios periféricos esclavos.

Sin embargo, la gran mayoría de las aplicaciones usan un MCU como maestro y este inicia el control de la transferencia de datos a los esclavos.

Para la comunicación se utilizan 4 líneas (*ver Gráfico 1.17*):

La Comunicación serial sincrónica trabaja bajo el esquema maestroesclavo y facilita la comunicación con dispositivos externos como LCD, teclados, conversores de señales analógicas a digital y memorias de interfaz serial, entre otros.

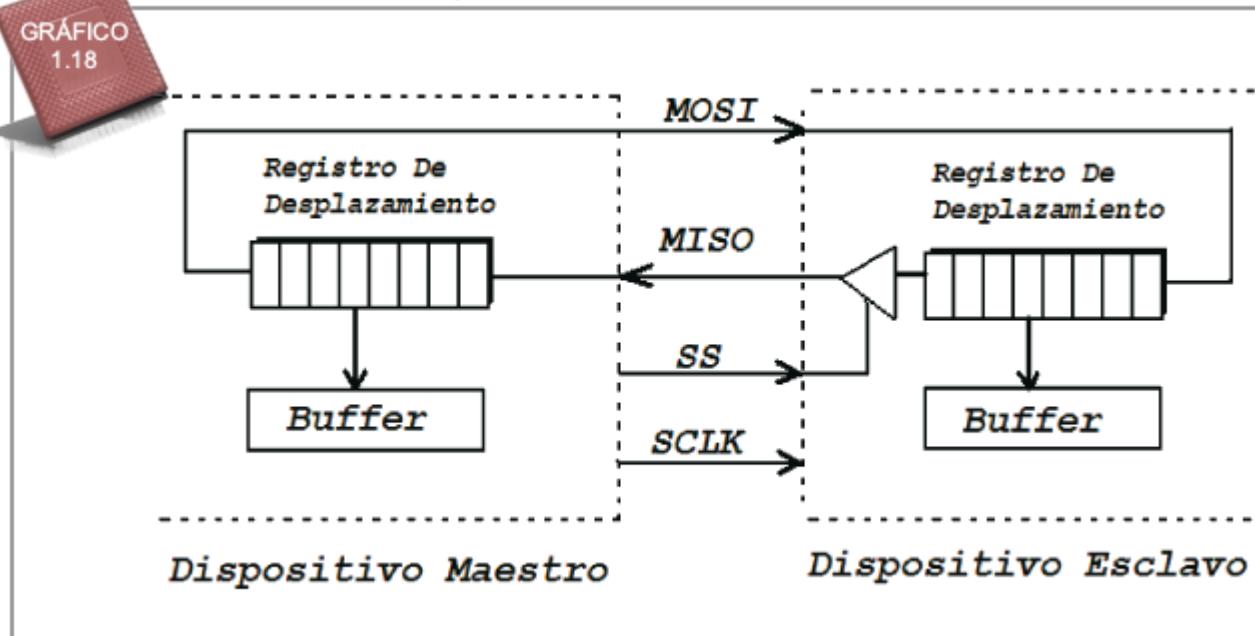
1. Señal de Selección de dispositivo **SS (Slave Select)**: en el maestro es una salida y en los esclavos es una entrada, que habilita mediante un nivel bajo (“0” lógico) el dispositivo con cual se desea establecer comunicación.
2. Señal de Datos del Maestro **MOSI (Master Output Slave Input)**: es una línea de datos que sale del maestro y entra a la línea de datos de los esclavos. Es la línea mediante la cual el maestro envía datos al esclavo seleccionado mediante la línea SS.
3. Señal de Datos del Esclavo **MISO (Master Input Slave Output)**: es una línea de salida en los esclavos y entrada en el maestro, mediante esta línea el esclavo seleccionado envía datos al maestro.
4. Señal de Sincronización **SCLK (Serial Clock)**: señal de salida en el maestro usada para realizar el muestreo de cada uno de los bits trasmitidos desde y hacia el maestro por las líneas MOSI y MISO (*ver Gráfico 1.18*).



En el Gráfico 1.18 se puede observar el diagrama básico del módulo SPI.



Movimiento de datos vía SPI.



El SPI permite realizar transmisiones Full duplex, de hecho siempre que sale un dato del maestro, se recibe también un nuevo dato en su buffer proveniente del esclavo, vale anotar que si el dato no es esperado este deberá ser ignorado.

La tasa de transferencia determinada por la velocidad de la señal SCLK, es configurable por el usuario del módulo, el cual, mediante registros, puede variar lo mismo que la polaridad (CPOL), o flancos en los cuales serán validados los datos, permitiendo flexibilidad al momento de elegir el dispositivo esclavo.

El sistema permite configurar notificación sobre los eventos en este módulo mediante interrupciones a la CPU y de esta forma facilitar el manejo y desempeño del sistema.

Los eventos básicos son:

- Registro de recepción del SPI lleno.
- Registro de transmisión del SPI vacío.

Este módulo permite y facilita la comunicación con dispositivos externos como son LCD, teclados, conversores de señales analógicas a digital, memorias de interfaz serial, entre otros dispositivos.

1.4.8 Comunicación serial I2C

El bus I2C (*Inter-Integrated Circuit*) fue desarrollado a principios de 1980 por la compañía Philips. Su propósito original fue proveer una forma sencilla de conectar un microcontrolador a chips periféricos en un televisor.

El I2C fue adoptado luego por varios fabricantes como un protocolo estándar de comunicación serial que permite transferencia de datos entre el microcontrolador y un dispositivo externo, el cual puede ser, entre otros, una memoria de datos seriales, un LCD, un sensor de temperatura, un conversor analógico a digital o un expansor de entradas y salidas. Esta interfaz está diseñada para operar máximo a 100 kbps y trabaja de forma similar al SPI en topología maestro-esclavos, con la ventaja de requerir menos líneas para la comunicación.

Para la implementación de esta comunicación se usan 2 pines denominados **SDA** (línea de datos bidireccional) y el **SCL** (línea de sincronización); todos los dispositivos conectados a estas líneas deben tener salidas de conexión “*open drain*” o de colector abierto que permiten al sistema el manejo de datos en ambos sentidos y la conectividad de varios dispositivos al bus. Las líneas deben contener externamente resistencias de *pull-up*, cuyo valor depende de las velocidades y de los dispositivos conectados.

El protocolo está conformado por 4 partes:

1. Señal de START.
2. Selección de dispositivo esclavo.
3. Transferencia de datos.
4. Señal de STOP.

La señal de START está definida como una transición de nivel alto a bajo de la línea SDA mientras el pin de SCL está en nivel lógico alto, esta señal denota el comienzo de una nueva transferencia de datos (cada transferencia puede contener varios bytes de datos) (ver Gráfico 1.19).

El siguiente byte después de la señal de START corresponde a la dirección del esclavo seleccionado por el maestro, contenido en el último bit la acción deseada sobre el esclavo:

1 → Lectura: configuración para que el esclavo trasmite datos al maestro

0 → Escritura: configuración para que el maestro trasmite datos al esclavo

Después de este byte, sólo el esclavo seleccionado responderá con una señal de reconocimiento (*acknowledge*), que lo hace enviando un nivel lógico cero por el pin SDA en el noveno bit de selección del esclavo.



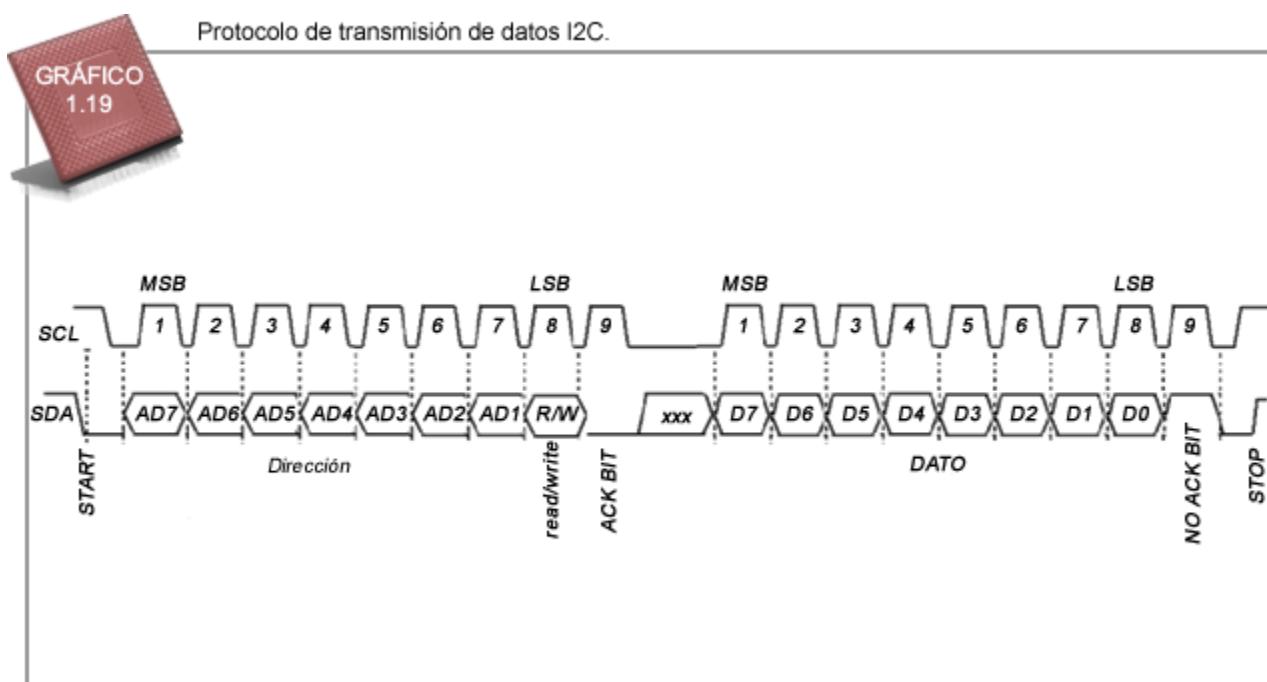
El bus I2C permite una Comunicación serial sincrónica, la cual consiste en el intercambio de datos entre un microcontrolador y un dispositivo externo, como un sensor, una memoria de datos seriales o un conversor analógico entre otros muchos.



Después de esta asignación, es iniciada la trama de datos byte por byte en el sentido especificado por el bit correspondiente de R/W. El cambio de cada bit solo puede ser generado mientras la línea de SCL está en bajo y deberá permanecer estable mientras la línea de SCL está en alto, como se muestra en el Gráfico 1.19. Existe un pulso en la línea SCL por cada uno de los bits de datos, y el MSB es transmitido primero. Cada byte de datos es seguido por un 9ºbit de reconocimiento (*acknowledge*), el cual es enviado por el dispositivo receptor de los datos.

Una vez enviada la secuencia de datos, el maestro termina la comunicación enviando una señal de STOP para liberar el bus de comunicación.

La comunicación está brevemente descrita en la siguiente figura:



El módulo I₂C puede además notificar a la CPU sobre la ocurrencia de varios eventos que permiten el manejo del envío de tramas de forma más sencilla y eficiente para el procesador.

1.5 Interrupciones en Sistemas Embebidos Microcontrolados

1.5.1 El Concepto de Interrupción

Una interrupción es un evento que notifica a la CPU sobre la ocurrencia de una situación excepcional de uno de sus periféricos. Tal como se sabe, la CPU está ejecutando instrucciones que corresponden a la labor normal de la aplicación o bien podría estar en algún modo de bajo consumo, mediante las instrucciones STOP o WAIT.

Para ilustrar el concepto de interrupción se utiliza tal cual el concepto de la vida real:

*Imagine una oficina, conformada por un **director** y varios **empleados** que trabajan en cubículos a su alrededor, cada uno realizando una función específica que debe llevar a cabo durante el día.*

Para el caso del **director** la tarea consiste en revisar una estrategia mensual: programación de tareas a cada uno de sus **empleados**, para lo cual usa un libro **de apuntes** donde va listando por empleado cada una de las tareas asignadas.

El trabajo del director se lleva a cabo de forma normal, pero de repente un empleado **golpea a la puerta**:

— “toc” “toc”.

El **director** que está escribiendo en su **libro de apuntes**, termina de escribir la frase que había empezado, y a continuación lanza la siguiente pregunta:

—**¿Quién toca a la puerta?**

El empleado responde:

—Soy Juan Carlos y es urgente.

Al ver esta situación el director accede a recibirlo, no sin antes tomar nota en una **hoja de apuntes varios** la tarea que está realizando, esto para evitar que una vez retome su trabajo luego de la interrupción olvide en cuál empleado estaba trabajando y la tarea que le estaba asignando.

El director abre la puerta para atender a Juan Carlos, pero también, y para evitar que otro empleado llegue a interrumpirlos, se asegura de colocar un aviso en la puerta que dice con claridad: “**No Interrumpir Temporalmente**”.

Juan Carlos entonces entrá a la oficina y el director le pregunta:

—**En qué puedo ayudar? Trata de ser breve porque se puede atrasar mi trabajo.**

Juan Carlos le pide al director que por favor pase por el depósito para que verifique que la mercancía que llegó es la correcta. El director se dirige al depósito, realiza la verificación y regresa nuevamente a la oficina a continuar con las tareas normales. Al entrar a la oficina remueve el letrero de “No Interrumpir Temporalmente”, ya que considera que la labor que le pidió Juan Carlos la realizó a cabalidad, y de esta forma los demás empleados sabrán que el director está ocupado pero eventualmente puede ser interrumpido de forma breve.

Es de anotar que el director en determinadas ocasiones puede poner el letrero de “No Interrumpir Temporalmente” sin necesidad de una interrupción; por ejemplo, cuando se encuentra realizando una labor muy crítica que requiere ser terminada con urgencia.

También es importante anotar que si estando en la reunión con Juan Carlos, los demás empleados pueden requerir alguna atención, para esto y dado que en la puerta se indica “No Interrumpir Temporalmente” ellos pueden dejarle una nota que ingresan debajo de la puerta, indicando por ejemplo: “Director: necesito hablar con usted urgente. Attn: Mario”. De tal forma que una vez el director llegue a la oficina lea el recado y pueda tomar la decisión de atender al Mario.

Consideré también que si las interrupciones ocurren con mucha frecuencia y si sumados los tiempos de las mismas resultan apreciables, las labores propias del director se atrasarán o pueden incluso no llegarse a realizar. Otro esquema que el director puede adoptar es no colocar el aviso de “No Interrumpir Temporalmente”, de modo que otros empleados podrían solicitar atención mientras el director está atendiendo a otro empleado, lo que además de confundirlo, lo llevaría a entretenérse en otras labores e incluso olvidar su labor principal, la cual podría no llevarse a cabo.

También es importante considerar que si el director está con Juan Carlos, pero de repente recibe una llamada que identifica como de su esposa, le podría decir a Juan Carlos que lo disculpe y tomar la llamada por considerarla de prioridad, al recordar que en la mañana ella llevó a su hijo Federico al médico y espera muy ansioso el resultado.

Para el esquema anterior el **director** hace el papel de **CPU**, **Juan Carlos** y **Mario** el de **interrupciones enmascarables**, y la **esposa del director** hace de **interrupción no enmascarable (SWI)** o de prioridad. El **libro de apuntes** del director es la **tarea principal**, y la **hoja de apuntes** varios equivale al **stack**.

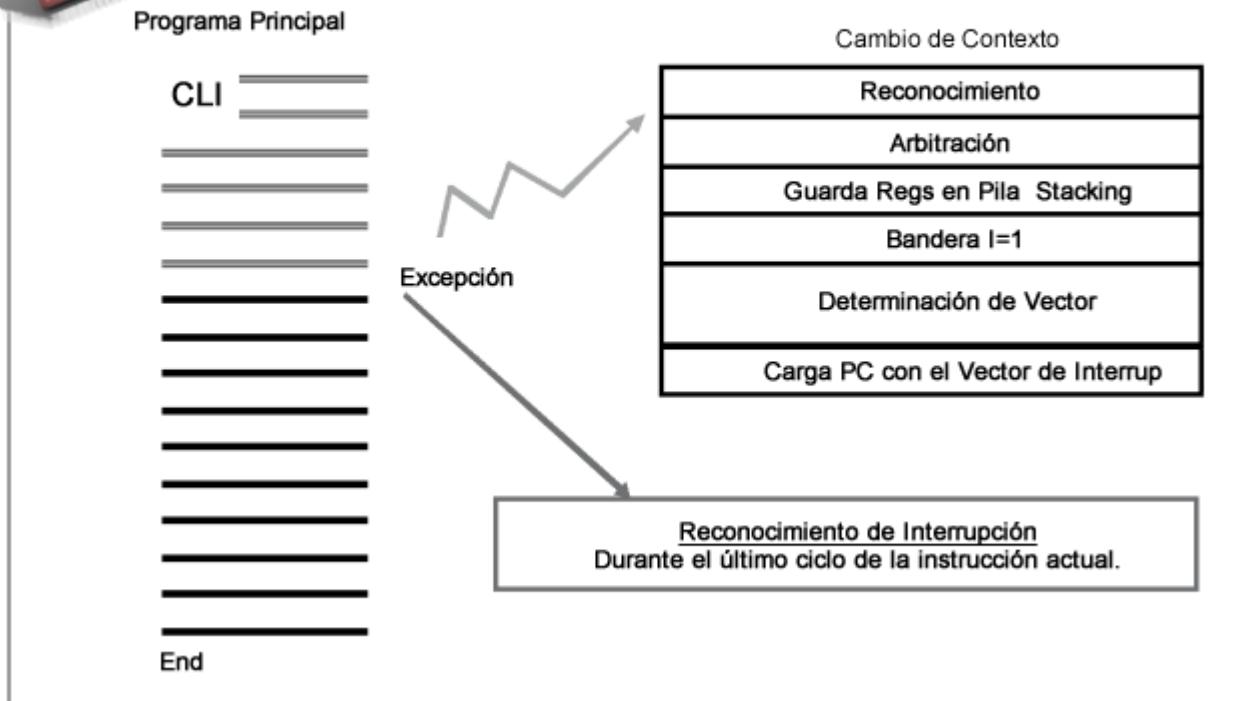
Ahora sí podemos ver la manera en que el procesador actúa ante una interrupción.

1.5.2 ¿Cómo trabaja el procesador ante una interrupción?

El procesador inicializa cada uno de los periféricos y ejecuta la función **EnableInterrupts()** que en ensamblador es la instrucción **CLI**, a continuación el procesador pasa a ejecutar su labor dentro del programa principal **main()** o una de sus funciones, y si uno de los periféricos genera una interrupción, la CPU termina la instrucción de ensamblador que esta ejecutando, en seguida realiza el **Reconocimiento** de la interrupción (*Gráfico 1.20*), el cual consiste en suspender la tarea principal y no continuar con la siguiente instrucción del programa principal.



Tratamiento de una interrupción de la CPU.



Luego continúa con la **Arbitración**, en la cual se determina la fuente de la excepción y su prioridad (el *Reset* se considera de mayor prioridad sobre cualquier interrupción y no requiere arbitraje).

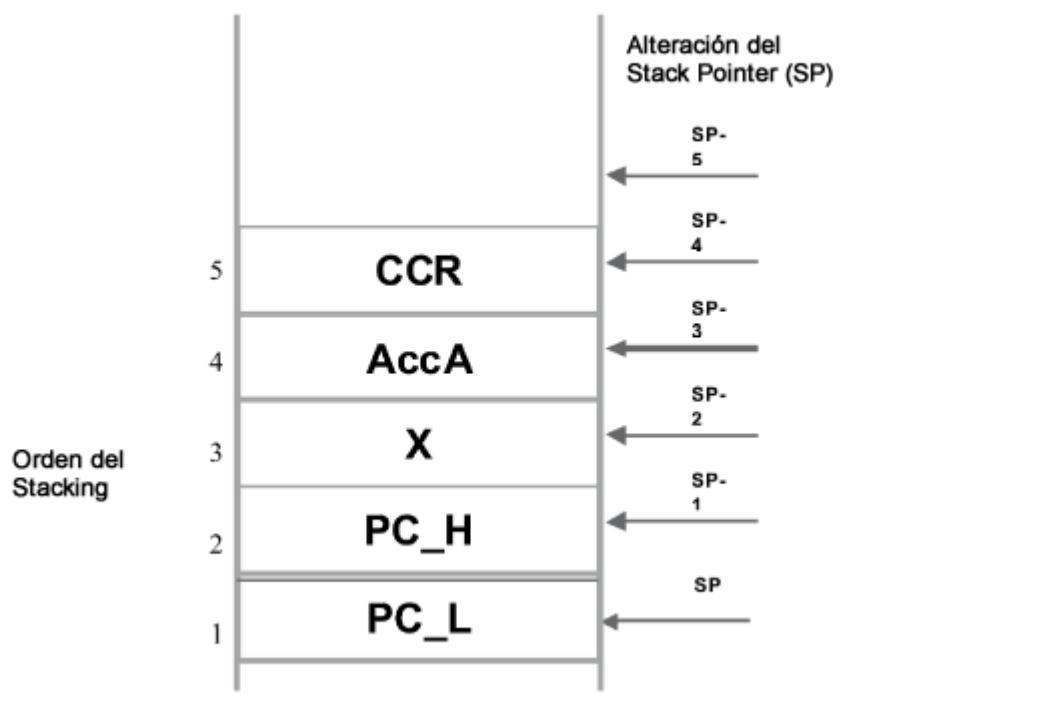
Si una o más interrupciones están pendientes, la interrupción de mayor prioridad será atendida primero y una vez sea tomada no existirá posibilidad que otra sea atendida, incluso si es de mayor prioridad. Este esquema NO permite las interrupciones anidadas.

En el guardado de registros en la pila (**Stacking**) se pone la bandera de I (Interrupción) en el CCR en 1, y se procede a enviar el modelo de programación del microcontrolador a la pila (*stack*) en el siguiente orden: PCL, PCH, X, AccA, CCR (ver Gráfico 1.21).

Nótese que el registro H no es puesto en el *stack*, la arquitectura HC(S)08 lo hace de esta forma para tener compatibilidad con su familia antecesora HC05, la cual no contenía registro H; sin embargo, este aspecto solo debe ser tenido en cuenta en programación en lenguaje ensamblador, porque en el compilador de Codewarrior® y la mayoría de los compiladores del mercado, el *stacking* del registro H se hace de forma automática una vez que la función es declarada de tipo interrupción (palabra reservada “*interrupt*”).



Orden de almacenamiento de registros en la pila ante una interrupción HC(S)08.



Cuando ocurre una interrupción el procesador atiende en primer lugar la interrupción de mayor prioridad, y atendida ésta no existe la posibilidad que otra sea tomada, incluso si es de mayor prioridad.

Pasará luego a la **Determinación del Vector** a partir de la tabla de interrupción (*Tabla 1.2*), en la cual decidió la fuente de la interrupción, y este valor (**Funcion_ISR**) lo **Transfiere al PC** de la CPU, donde continua con la ejecución de la rutina de interrupción que solicitó su atención.



Vectores de interrupción del microcontrolador AP16A.

TABLA
1.2

Dirección Vector	Interrupción	Prioridad	L
0xFFD0 – 0xFFD1	Reserved – No usado	23	
0xFFD2 – 0xFFD3	Timebase – Base de Tiempo	22	
0xFFD4 – 0xFFD5	Ir SCI Tx – Transmisión Ir SCI	21	
0xFFD6 – 0xFFD7	Ir SCI Rx – Recepción Ir SCI	20	
0xFFD8 – 0xFFD9	Ir SCI Err – Error en Ir SCI	19	
0xFFDA – 0xFFDB	SPI Tx – Transmisión SPI	18	
0xFFDC – 0xFFDD	SPI Rx – Recepción SPI	17	
0xFFDE – 0xFFDF	ADC - Conversión Completa	16	
0xFFE0 – 0xFFE1	Keyboard – Pines Puerto D	15	
0xFFE2 – 0xFFE3	SCI Tx – Transmisión de SCI	14	
0xFFE4 – 0xFFE5	SCI Rx – Recepción de SCI	13	
0xFFE6 – 0xFFE7	SCI Err – Error de SCI	12	
0xFFE8 – 0xFFE9	MMIIC – Serial I2C	11	
0xFFEA – 0xFFEB	TIM2 Ov – Timer2 Sobre flujo	10	
0xFFEC – 0xFFED	TIM2 Ch 1 – Timer2 Canal 1	9	
0xFFEE – 0xFFEF	TIM2 Ch 0 – Timer2 Canal 0	8	
0xFFF0 – 0xFFF1	TIM1 Ov – Timer1 Sobre flujo	7	
0xFFF2 – 0xFFF3	TIM1 Ch 1 – Timer1 Canal 1	6	
0xFFF4 – 0xFFF5	TIM1 Ch 0 – Timer1 Canal 0	5	
0xFFF6 – 0xFFF7	PLL – Ajuste del Oscilador	4	
0xFFF8 – 0xFFF9	IRQ2 – Pin Externo 2	3	
0xFFFFA – 0xFFFFB	IRQ1 – Pin Externo 1	2	
0xFFFFC – 0xFFFFD	SWI – Interrupción de Software	1	
0xFFFFE – 0xFFFFF	Reset	0	H

Funcion_ISR



Ejecución de la función de interrupción → Funcion_ISR.

Función_ISR :Nombre de la función
PSHH :Almacena el registro H en el Stack
 :Comentarios
 :
 :
PULH :Recupera el registro H del Stack
RTI :Retorno de la ISR

Retorno de interrupción a la aplicación principal:

Una vez en la rutina de interrupción el control de la aplicación es completo de la función, al terminar su labor, el proceso de retorno deberá recuperar el registro H, mediante la instrucción PULH (*Gráfico 1.22*), y regresar al programa principal mediante la instrucción RTI (*Return From Interrupt*). Estas dos instrucciones las adiciona el compilador de C una vez el programador ha creado la función como rutina de interrupción.

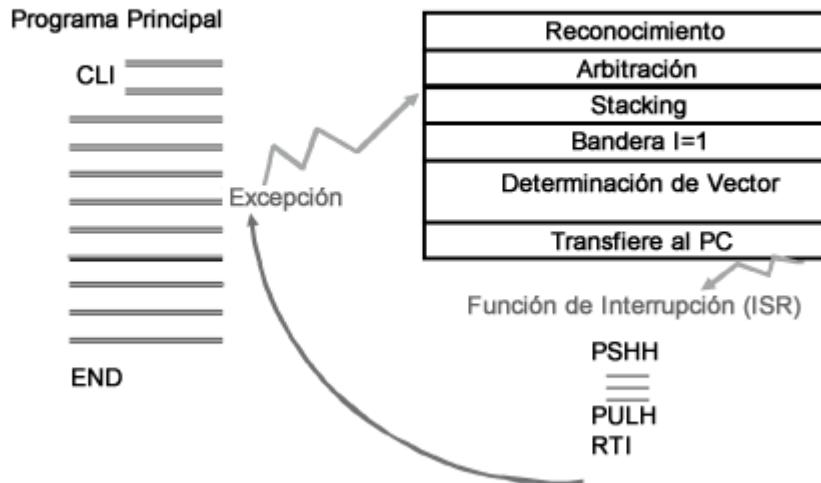
La instrucción **RTI** (*Gráfico 1.23*), hace que se recupere el contexto anterior y el programa continúa en el lugar en el cual fue interrumpido.



Una vez atendida la interrupción, la instrucción RTI permite volver al contexto anterior, es decir, al programa principal.



Retorno de interrupción a la aplicación principal.



1.6 CAMBIO DE CONTEXTO

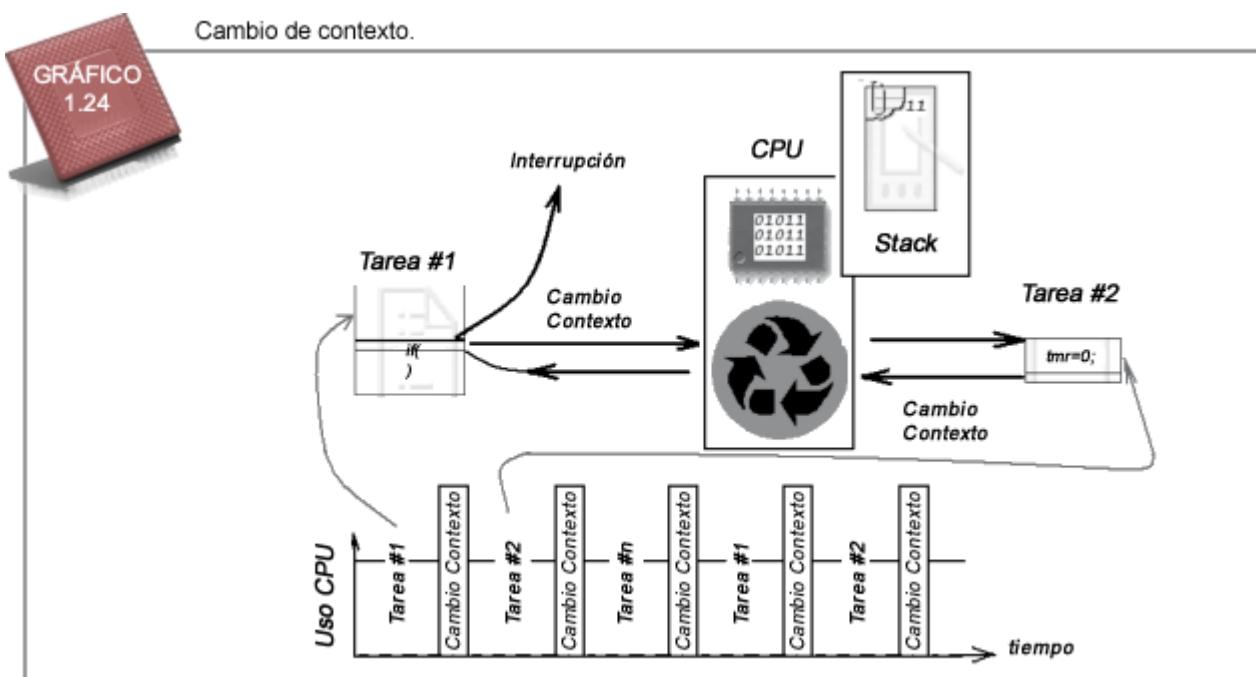
Una interrupción también permite que el procesador se ocupe de un nuevo proceso, dejando el anterior en estado suspendido en la RAM; dicha acción se conoce como Cambio de contexto y es un elemento clave en el desarrollo de sistemas operativos.

Se denomina **cambio de contexto** (*context switching*) al proceso de suspender la ejecución normal de un programa, guardar el estado del modelo de programación en RAM y ejecutar otro proceso para iniciar o continuarlo.

Un ejemplo de cambio de contexto se da a causa de una interrupción al procesador central CPU, en la cual se requiere que la CPU y el modelo de programación ejecuten un procedimiento especial denominado función de interrupción.

Se presenta un nuevo cambio de contexto al terminar la rutina de interrupción (con la instrucción RTI), en la cual el modelo de programación es recuperado del *stack* y se continúa el proceso que fue suspendido.

El cambio de contexto es la base de los sistemas operativos de tiempo real, el cual consiste en ejecutar diferentes tareas de forma periódica, lo que hace que cada tarea realice su función (ver Gráfico 1.24).



1.7 LATENCIA DE INTERRUPCIÓN



Cada vez que el procesador recibe un llamado de interrupción, ha de tomar un tiempo en responder a él; este lapso es la Latencia de interrupción, y lo ideal es que el programador las desarrolle en rutinas cortas y eficientes. Esto aumentará la velocidad del sistema para responder a nuevas interrupciones.

Debido a que las interrupciones son una forma de mejorar la respuesta de un sistema, y la rapidez a esta respuesta es importante, una pregunta lógica es: ¿Qué tan rápido responde el sistema ante una solicitud de interrupción?, en realidad depende de varios aspectos:

1. El máximo tiempo que permanecen las interrupciones deshabilitadas.
2. El tiempo de ejecución que toma la rutina de interrupción o interrupciones de mayor prioridad que la interrupción en cuestión.
3. El tiempo que le toma al procesador terminar la instrucción que está ejecutando, realizar el *stacking* e iniciar la ejecución de la primera instrucción de la interrupción.

Se denomina Latencia de Interrupción (*interrupt latency*) al tiempo que le toma a un sistema responder al llamado de una interrupción.

El factor 3 depende básicamente del fabricante de procesador, y no está bajo el control del programador; sin embargo, se

pueden controlar los factores 1 y 2, haciendo las rutinas de interrupción lo más cortas y eficientes posible. Adicionalmente, en el programa general se debe procurar deshabilitar las interrupciones por lapsos cortos de tiempo y solo cuando sea exclusivamente necesario.

1.8 ZONAS CRÍTICAS DE SOFTWARE



Hay momentos en que el programa principal está desarrollando un proceso crítico que no puede ser interrumpido. En esos casos, el programador dejará zonas del código con las interrupciones deshabilitadas, llamadas Zonas críticas de software.

Existen zonas dentro del programa principal en las cuales deben ser deshabilitadas algunas, o todas las interrupciones, esto con el fin de evitar que la aparición de una interrupción perturbe un procedimiento de datos o de hardware que el procesador esté ejecutando en el programa principal. La sección de código en la cual las interrupciones están deshabilitadas por no poder ocurrir, se denomina una *zona crítica de software*.

Son ejemplos de zona crítica el manejo de tiempo por *polling* o por ciclos de reloj, el manejo de estructuras y listas de datos compartidos que sean modificadas dentro y fuera de interrupciones, códigos de programa que deban ejecutarse de forma atómica, ejecuciones críticas del software que deban cumplir con un tiempo determinado en ejecución.



Receta

Es responsabilidad del programador identificar sus zonas críticas en el software, para deshabilitar las interrupciones y habilitarlas de nuevo, una vez que se ha terminado de ejecutar el código crítico.

```
DISABLE_INTERRUPTS ;
```

....

Zona Critica de Software

....

```
ENABLE_INTERRUPTS ;
```

1.9 Herramientas para Diseño Embebido en el mercado

1.9.1 Herramientas para Microchip



PICSTART Plus

Es una herramienta que permite la programación de los microcontroladores de Microchip en empaque DIP, se conecta vía serial al PC con el software integrado de desarrollo MPLAB IDE incluido en el costo del programador.

El sistema incluye una copia libre del compilador PICC Lite por un costo de USD 200.00 (precio FOB² EE.UU.) bajo el número de parte **DV003001**.

Se puede descargar una versión libre del compilador C30 de Microchip de la página: <http://www.microchip.com/c30>

MPLAB PM3



Es un programador universal de fácil operación que permite la programación en serie o en producción, ya sea conectada a un PC o en modo independiente (*standalone*), sin la necesidad de un computador.

Soporta la programación de la familia PICs, dsPIC^R DSC (*Digital Signal Controller*). El sistema se puede actualizar para que soporte nuevos dispositivos. Su costo en el mercado es de USD 895.00 (precio FOB EE.UU.) bajo el número de parte **DV007004**.

MPLAB ICD 2



Es una herramienta de bajo costo, soporta programación y depuración en tiempo real de los microcontroladores PIC, de los dsPIC^R DSC.

Los programas son enviados a la herramienta, por medio de funciones del MPBLAB ICE, permite la visualización de variables, poner puntos de ruptura (*breakpoints*) de programas desarrollados en C o en código ensamblador.

El costo de la herramienta es de USD 160.00 (precio FOB EE.UU.) bajo el número de parte **DV164005**.

MPLAB REAL ICE



Es un sistema emulador de última generación para los procesadores DSC y MCUs de Microchip, realiza depuración y programación de forma rápida, de fácil manejo y en un ambiente gráfico muy amigable.

La conexión al PC es vía USB 2.0 de alta velocidad y se conecta al “*target*” ya sea con conector RJ11 o con el conector especializado para ruido LVDS (*low voltage differential signal*).

Es igualmente actualizable con las versiones posteriores del MPLAB IDE, nuevos dispositivos y nuevas características.

Es el sistema de desarrollo más recomendado por Microchip en términos de inversión y de características. Su precio es de USD 499.00 (precio FOB EE.UU.), bajo el número de parte **DV244005**.

ICD-U40



Esta interfaz de hardware se conecta entre el PC a través del puerto USB y la aplicación que contiene el microcontrolador PIC, usando 2 de sus pines. Permite la programación y la depuración en circuito, sin remover el microcontrolador de la aplicación, brinda de forma opcional brindar alimentación al microcontrolador y a la aplicación.

Esta herramienta es la sugerida cuando el compilador usado sea el de la misma compañía CCS, porque integra de forma robusta el manejo de los ambientes de programación PCW, PCWH, PCDHD y PCDIDE, mas no es soportada por el popular ambiente MPLAB®IDE.

La herramienta fue desarrollada por la compañía CCS y se consigue bajo el número de parte ICD-U40 a un precio de USD 75 (precio FOB EE.UU.).



MPLAB ICE 2000

Es un sistema emulador de muy alta velocidad (30 MHz) que soporta bajo voltaje de operación 65,535 puntos de ruptura, análisis de trazas complejas (*trace*) para la gran mayoría de los MCUs de Microchip; sus diferentes adaptadores intercambiables permiten al sistema configurarlo para soportar diferentes procesadores.

Permite la captura en tiempo real de valores en registros, en RAM y en periféricos, sin necesidad de detener la ejecución del procesador.

Su costo en el mercado es de USD 1,000.00 (precio FOB EE.UU.) bajo el número de parte **ICE2000**.

-
- 2 FOB (Free on Board): es el precio de venta de los bienes embarcados a otros países, puestos en el medio de transporte sin incluir valor de los seguros y fletes.

1.9.2 Herramientas para Renesas

H8/Tiny Starter Kit Plus



El SKP es un sistema completo de evaluación para la serie H8/300H uTINY de los microcontroladores de la compañía Renesas. El kit contiene una tarjeta de evaluación y un emulador. El software de desarrollo es llamado el HEW (*High-performance Embedded Workshop*) que permite desarrollo y depuración en lenguaje C con una capacidad de hasta 64K de código compilado. El producto es ordenado bajo el número de parte SKPH8TINY con un costo de USD 119.00 (precio FOB EE.UU.).

1.9.3 Herramientas para Texas Instruments



eZ430-RF2500

Consiste en una herramienta pequeña y de bajo costo que se conecta vía USB al PC. Provee el software y hardware para evaluar los microcontroladores MSP430F2274 y además el transceiver CC2500 de 2.4 GHz. Para la serie MSP430F2013 se cuenta con una herramienta similar con número de parte eZ430-F2013.

Para el software utiliza el IDE *IAR Embedded Workbench* que permite escribir, bajar el código al micro y depurarlo, así como tener la aplicación ejecutándose a velocidad completa con varios puntos de ruptura. Su costo en el mercado es de USD 49.00 (FOB EE.UU.).

1.9.4 Herramientas para Freescale



Spyder

Es una pequeña y sencilla herramienta de programación y depuración vía BDM (*Background Debug Mode*), del tamaño de una memoria USB. Fue desarrollada por la compañía SofTec Microsystems y es soportada por Freescale™.

Spyder soporta los procesadores de la subfamilia **MC9S08QG**, **RS08KA** y **MC9S08QD**.

La herramienta viene con un CD el cual incluye la última versión de Codewarrior® para HC(S)08, el manual de usuario y toda la documentación necesaria para instalación y puesta en marcha.

El precio en el mercado esta alrededor de USD 29.00 (precio FOB EE.UU.) y está bajo el número de parte: **USBSPYDER08**



Multilink

Este es un interfaz USB a BDM de bajo costo y fácil implementación que soporta todos los microcontroladores miembros de los cores HCS08, HC12, HCS12 y ColdFire V1 de Freescale™. Con lo que desarrollar desde 8 bits hasta 32 bits resulta transparente para el programador debido a que se usa el mismo conector de 6 pines BDM, y el software para todos, integrado Codewarrior® 6.0 o superior.

El programador puede hacer depuración en tiempo real, borrar y programar la flash, ya sea una tarjeta de evaluación o en el sistema final de aplicación, lo cual permite que a una aplicación se le pueda actualizar su firmware, solo conectando el Multilink y realizando la descarga del software.

Resulta bastante práctica ya que permite hacer depuración en la aplicación permitiendo poner puntos de ruptura (*breakpoints*), que facilitan conocer el trayecto y operación del software.

Contiene además varios niveles de disparo de depuración basados en la ejecución del programa y/o eventos que permite tener un historial sobre la ejecución antes de llegar al punto ruptura, que da al programador información más detallada sobre el funcionamiento del software.

El costo en el mercado es de USD 99.00 (precio FOB EE.UU.) y puede ser ordenada bajo el número de parte **USBMULTILINKBDME**.

CyclonePro



Este sistema permite, además de las funcionalidades del Multilink, la posibilidad de trabajar independiente de un PC, lo que le otorga más versatilidad cuando se está en campo o en programación masiva de *firmware*, ya que solo requiere que el software sea enviado una única vez desde el PC, y luego, este puede fácilmente reproducirlo en los procesadores finales.

Soporta todos los procesadores que se programan vía MON08, es decir HC08, y también vía BDM, es decir HCS08, HC12, HCS12 y ColdFire V1.

Contiene una serie de botones y leds y en algunos casos un display de cristal líquido el cual permite controlar la operación independiente “*standalone*” de programación.

Los leds indican al usuario si las opciones elegidas de borrado, programación, verificación resultaron exitosas o si requieren ser revisadas.

El sistema puede además proveer la alimentación al procesador a partir de su fuente interna y de manera inteligente si el procesador trabaja a 5 voltios o a 3 voltios.

La conexión al PC puede ser vía USB o vía Ethernet.

Tiene la opción de ser usada como interfaz para hacer depuración, programación de los procesadores soportados, aunque su orientación es más para programación en masa en una línea de producción o para programación y actualización de *firmware* en el campo, con lo que no se requiere de un PC y un manejo experto de software. El precio de la herramienta en el mercado es de USD 499.00 (precio FOB EE.UU.).

DemoQE128



Es una herramienta de bajo costo diseñada para demostrar, evaluar y hacer depuración principalmente de los procesadores MC9S08QE128 y MCF51QE128 de Freescale™. Dentro de esta herramienta está incluido todo el circuito del Multilink el cual permite tener el mismo manejo que se tiene con dicha herramienta. Cuenta con 4 entradas, 8 salidas digitales, 1 entrada analógica y la inclusión de un acelerómetro, con lo que el programador puede realizar prototipos, demostraciones y

evaluación de código, el cual es programado directamente en el procesador.

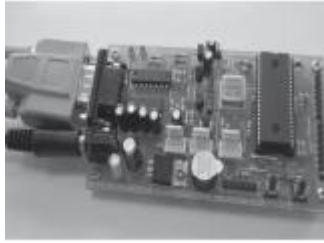
Contiene además acceso a todos los pines del procesador en los que se puede realizar conexión de periféricos externos, y acople de señales que vienen o van a la aplicación final.

La herramienta puede usarse también para programar las familias de microcontroladores HCS08, HC(S)12 y ColdFire V1 a través del conector BDM disponible para tal fin.

El costo en el mercado es de USD 99.00 (precio FOB EE.UU.) y puede ser ordenada bajo el número de parte **DEMOQE128**.

1.9.5 Herramientas utilizadas en los ejemplos

AP-Link Freescale APX



Este sistema de evaluación fue creado exclusivamente para ilustrar todos los ejemplos de este libro, permite que los programas compilados puedan ser programados en un microcontrolador Freescale™ y ejecutados de forma remota usando el depurador de Codewarrior®, o bien de forma independiente usando solo una fuente de alimentación estándar.

La evaluación de los ejemplos es opcional lo mismo que el ensamblaje de la tarjeta AP-Link, de tal forma que el lector puede realizar su aprendizaje sin necesidad de la tarjeta o realizar depuración con ella, dado que lo puede hacer en modo simulación completa con PC o “**Full Chip Simulation**” en todos los casos. Sin embargo, resulta bastante útil para las prácticas relacionadas con hardware externo al microcontrolador, refuerza bastante el entendimiento del lenguaje y su relación con el hardware en sistemas embebidos.

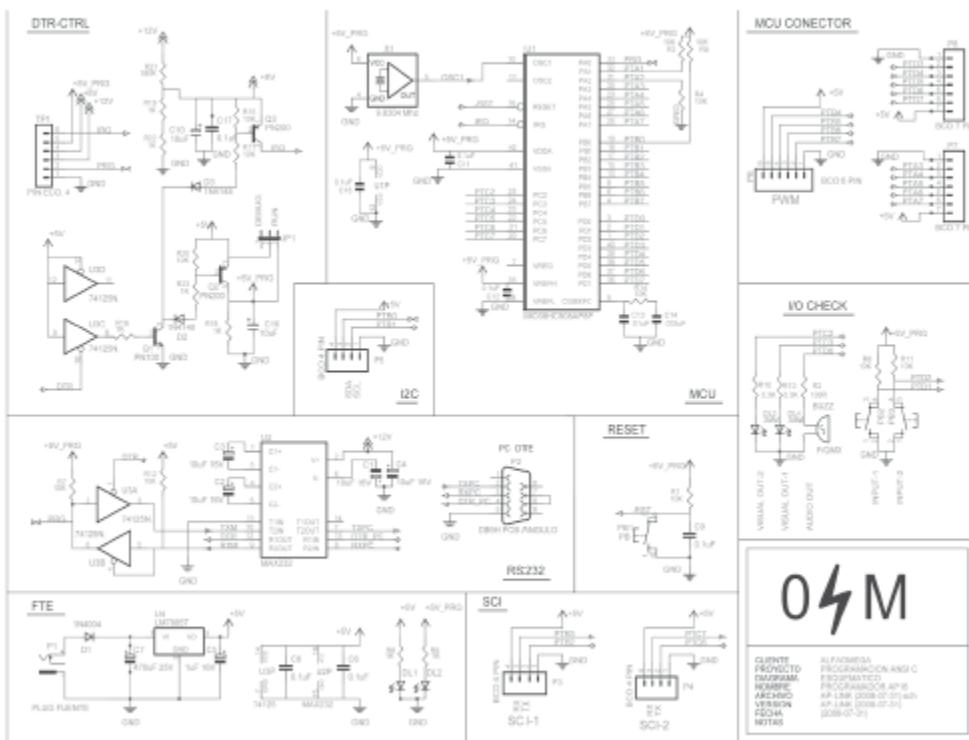
Si el lector solo está interesado en aprender la parte teórica del ANSI C y solo considera hacer las prácticas en modo simulación, podrá pasar directamente al Capítulo 2, si por el contrario, desea afianzar mejor sus conocimientos, el ensamblaje y prueba de la tarjeta es lo más recomendado.

Montaje del Sistema AP-Link

El montaje del sistema de evaluación incorpora varios pasos y elementos, los cuales están descritos a continuación:

**GRÁFICO
1.25**

Diagrama esquemático del Sistema de evaluación AP-Link.



Listado de Partes del Sistema de Evaluación AP-Link

Las partes requeridas para el ensamblaje del sistema de evaluación pueden adquirirse fácilmente en el mercado local de partes electrónicas. A continuación se encuentra el listado de partes con descripción y número de parte de uno de los proveedores de partes al detalle mundial DIGIKEY www.digikey.com, con el cual el diseñador podrá ordenar las partes para el ensamblaje.

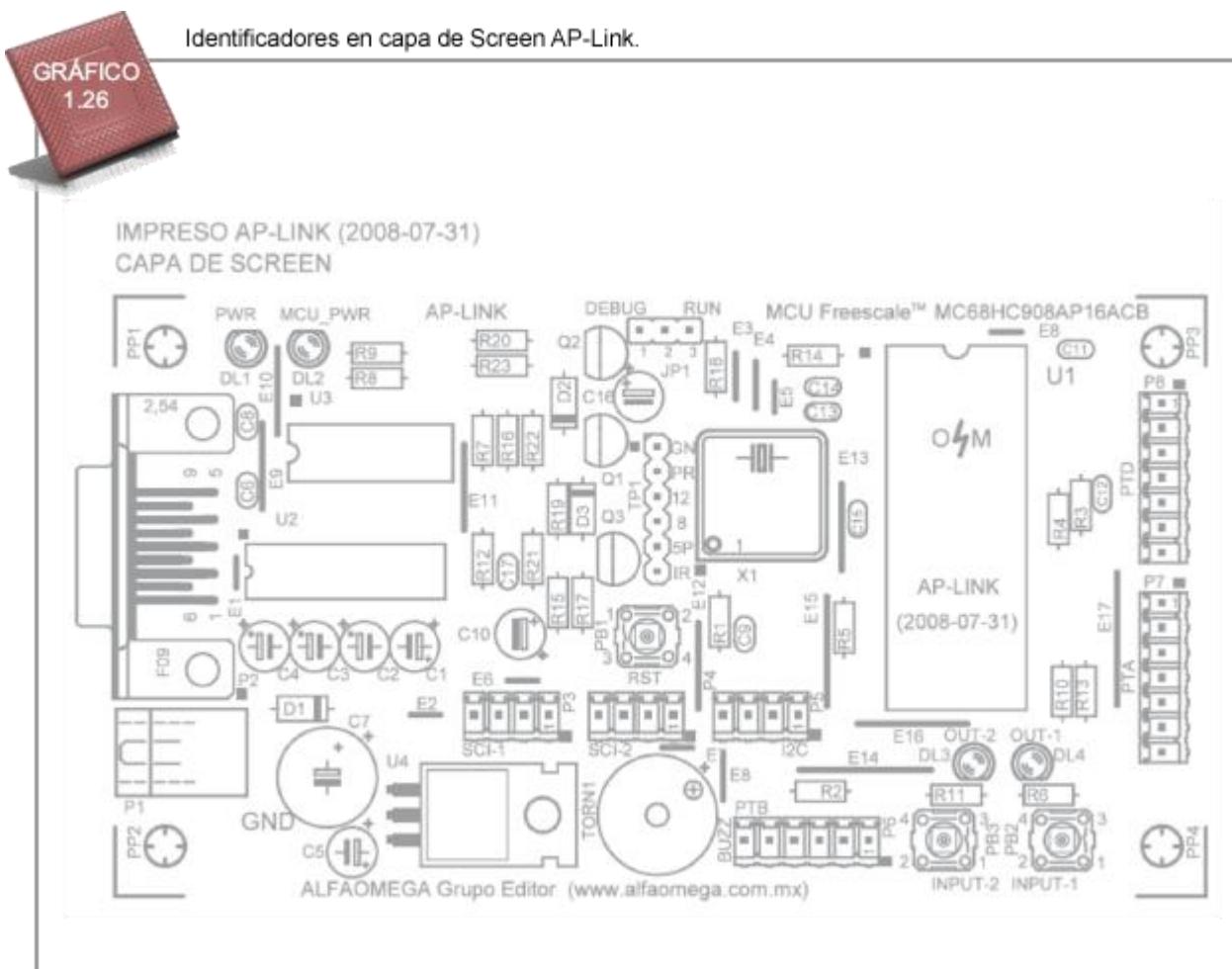
**TABLA
1.3**

Listado de partes del sistema de desarrollo AP-Link.

QTY	IDENTIFICADOR SCREEN AP-Link	DESCRIPCIÓN Y VALOR	DIGIKEY P/N
1	BUZZ	Buzzer 5 voltios Redondo	458-1061-ND
6	C1,C2,C3,C4,C10,C16	Condensador Aluminio 10uF 50V radial	P5567-ND
1	C5	Condensador Aluminio 1uF 50V radial	P5563-ND
7	C6,C8,C9,C11,C12,C15,C17	Condensador Cerámico monolítico 0.1uF 50V radial	399-4329-ND
1	C7	Condensador Aluminio 470uF 25V radial	P5574-ND
1	C13	Condensador Cerámico .01uF 50V radial	399-4327-ND
1	C14	Condensador Cerámico .033uF 50V radial	399-4363-ND
1	D1	Diodo 1N4004	1N4004DICT-ND
2	D2,D3	Diodo 1N4148, DO35	1N4148FS-ND
2	DL1,DL2	Diodo LED 3MM Rojo	516-1291-ND
2	DL3,DL4	Diodo LED 3MM Verde	516-1293-ND
5	E1,E2,E5,E6,E8	Puente Corto Circuito de 4.3 CM	NA
3	E3,E4,E7,E18	Puente Corto Circuito de 6.0 CM	NA
5	E9,E19,E13,E11,E12	Puente Corto Circuito de 10.0 CM	NA
2	E15,E16	Puente Corto Circuito de 12.3 CM	NA
2	E14,E17	Puente Corto Circuito de 14.7 CM	NA
1	FTE	Adaptador de voltaje AC/DC 9VDC 0.66 ^a	T980-P5P-ND
1	GND	Punta de prueba negro	5001K-ND
1	JP1	Jumper eléctrico 3 pines	WM8085-ND
1	JP1H	Jumper Hembra con agarradera	66464-102-ND
1	P1	Plug Fuente para PCB pin interno pequeño	CP-202A-ND
1	P2	Conector DB9 para PCB Ángulo de 90 grados	A35107-ND
3	P3,P4,P5	Conector Blanco 4 Pines	WM4202-ND
3	P6	Conector Blanco 6 Pines	WM4204-ND
2	P7,P8	Conector 7 pines macho	WM2705-ND
3	PB1,PB2,PB3	Pulsador 4 pines para PCB 6mm	P12192S-ND
4	PP1,PP2,PP3,PP4	Soportes plásticos	NA
1	Q1	Transistor PN100 o 2N2222 TO92	PN100A-ND
2	Q2,Q3	Transistor PN200 o 2N2907 TO92	PN200A-ND
12	R1,R3,R4,R5,R6,R7,R11,R12, R14,R15,R17, R20	Resistencia 10K Ohm @ 1/4W	10KQBK-ND
1	R2	Resistencia 100 Ohm @ 1/4W	100QBK-ND
7	R8,R9,R16,R18,R19, R22,R23	Resistencia 1K Ohm @ 1/4W	1.0KQBK-ND
2	R10,R13	Resistencia 3.3K Ohm @ 1/4W	3.3KQBK-ND
1	R21	Resistencia 180 Ohm @ 1/4W	180QBK-ND

6	TP1	Puntas de prueba para PCB	5000K-ND
1	TORN1	Tornillo 1/8", 8 mm de largo con arandela y tuerca	NA
1	U1	Base de 42 pines TH, SDIP 0.600	ED22426-ND
1	U1	Microcontrolador MC68HC908AP16ACB	MC908AP16ACBEND
1	U2	Integrado MAX232 16 pines DIP, R2232 Transceiver	MAX232ACPE-ND
1	U3	Integrado 74HC125N 14 pines DIP, Quad Buffer	MM74HC125N-ND
1	U4	Regulador de voltaje LM7805T TO- 220	LM7805ECT-ND
1	X1	Oscilador 4 pines 9.8304 MHz Half Size	XC266-ND

Capa de Screen:



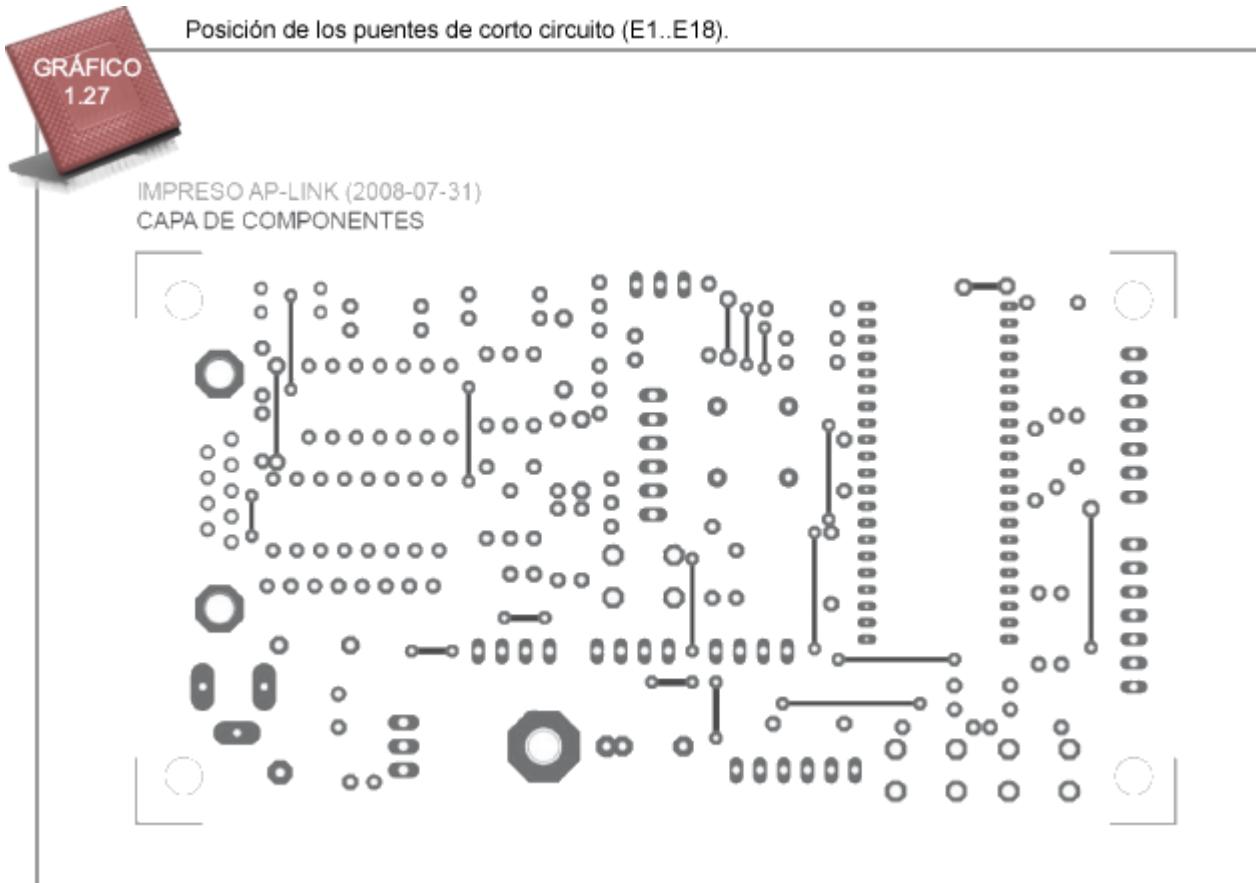


Capa de Componentes:

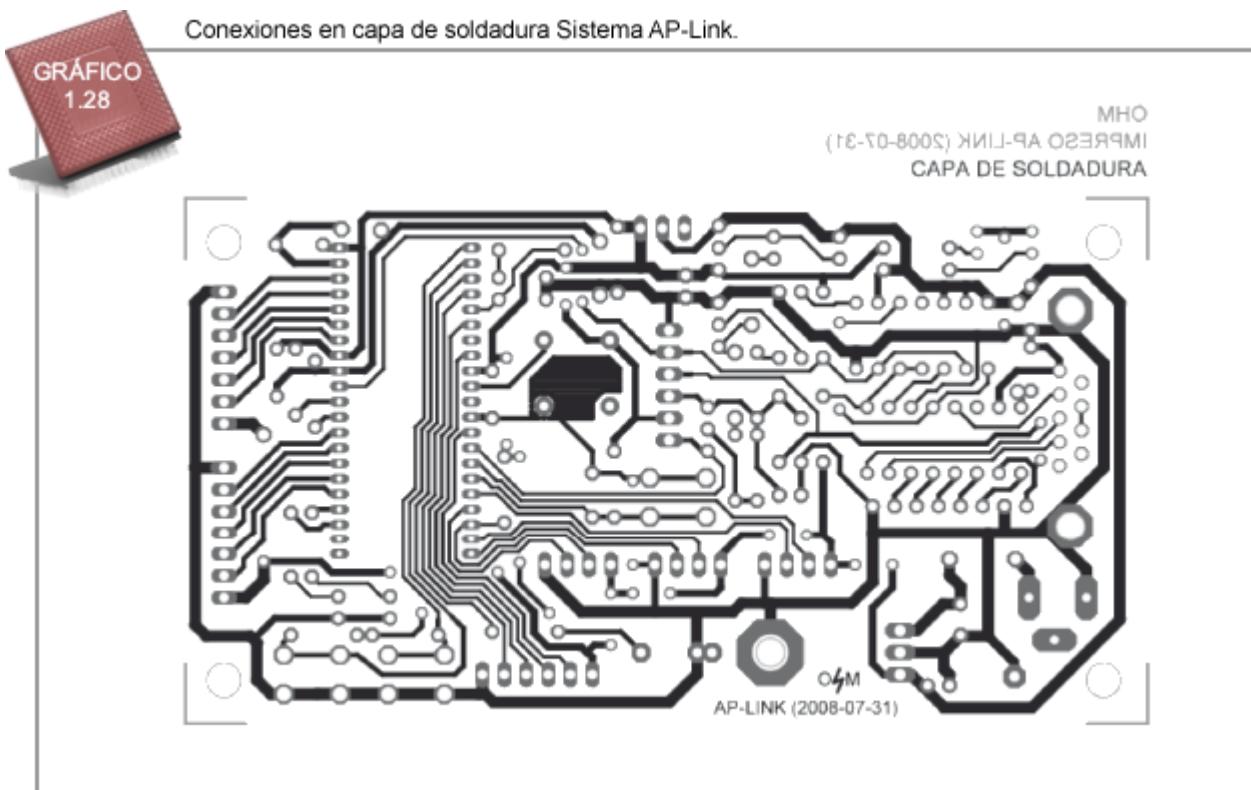
En esta capa se muestra la posición de los puentes corto circuitos descritos en la lista de partes:

E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,E11,E12,E13,E14,E15,E16,E17, E18.

La limpieza es fundamental al momento de ensamblar el sistema de evaluación. Se puede limpiar el impreso (ya armado), con algún limpiacontacto como el alcohol isopropílico.



Capa de Soldaduras:



Ensamble del Sistema AP-Link

Para el ensamble del sistema de evaluación se debe tener en cuenta que el PCB es de una sola capa y existen 18 conexiones (E1...E18) en la capa de componentes (*Gráfico 1.27*), los cuales deben ser ensamblados usando primero los *puente corto circuito*. Se recomienda luego continuar con el ensamble de las resistencias y condensadores por ser los componentes más bajos, luego el regulador de voltaje y la base del microcontrolador AP y por último los conectores.

Es importante limpiar el impresio ya armado con algún limpia-contactos, se recomienda usar alcohol isopropílico para este proceso.

Una vez ensamblada la tarjeta se requiere remover el microcontrolador **AP** del sistema, y proporcionar alimentación a la tarjeta con el adaptador de voltaje.

Realizar las siguientes medidas con un voltímetro con referencia **GND en pin 1 de TP**:

Con Jumper en Modo Run, y punta positiva en **Pin 4 de SCI-1 (+5V)**, la medida deberá ser de 5 voltios DC +- 5%. El Led DL1 deberá estar encendido, y el Led DL2 apagado.

Pin 3 de TP (+12V), la medida deberá ser entre +9 y +11 voltios.

Poner Jumper en modo RUN, el Led DL2 deberá encender. Medir el voltaje en **pin 9** de la base de **U1 (MCU)**, deberá ser de +5 voltios.

Pin 1 de U1 (MCU), deberá ser de +5 voltios, al presionar PB3 (INPUT-2), la medida deberá ser de 0 voltios.

Pin 2 de U1 (MCU), deberá ser de +5 voltios, al presionar PB2 (INPUT-1), la medida deberá ser de 0 voltios.

Pin 10 de U1 (MCU), deberá ser alrededor de +2.2 voltios. De lo contrario revisar conexión del oscilador.

Pin 16 de U1 (MCU), deberá ser alrededor de +5 volts, al presionar PB1 el voltaje deberá ser de 0 voltios.

Pin 19 de U1 (MCU), deberá ser de +5 voltios.

Pin 32 de U1 (MCU), deberá ser de +5 voltios.

Pin 33 de U1 (MCU), deberá ser de +5 voltios.

Pin 35 de U1 (MCU), deberá ser de +5 voltios.

Pin 42 de U1 (MCU), deberá ser de +5 voltios.

Posicionar la referencia **GND** del voltímetro en el **pin 4 de SCI-1 (+5V)** y realizar las siguientes medidas:

Pin 12 de U1 (MCU), deberá ser de -5 voltios.

Pin 31 de U1 (MCU), deberá ser de -5 voltios.

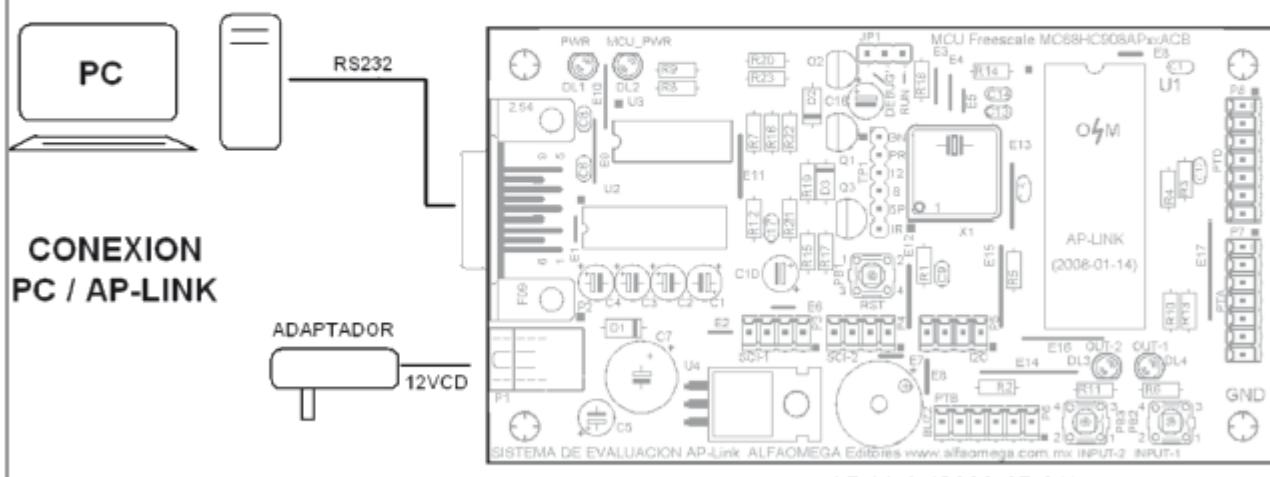
Pin 34 de U1 (MCU), deberá ser de -5 voltios.

Pin 41 de U1 (MCU), deberá ser de -5 voltios.

La prueba final de la tarjeta se realiza pasando el Jumper a DEBUG, insertando el microcontrolador AP en la base U1 y ejecutando la depuración de el ejemplo 1 en el capítulo 2 o **El Primer Programa en ANSI C** que se verá en el Capítulo 3.



Conexión del sistema de desarrollo AP-Link al PC.

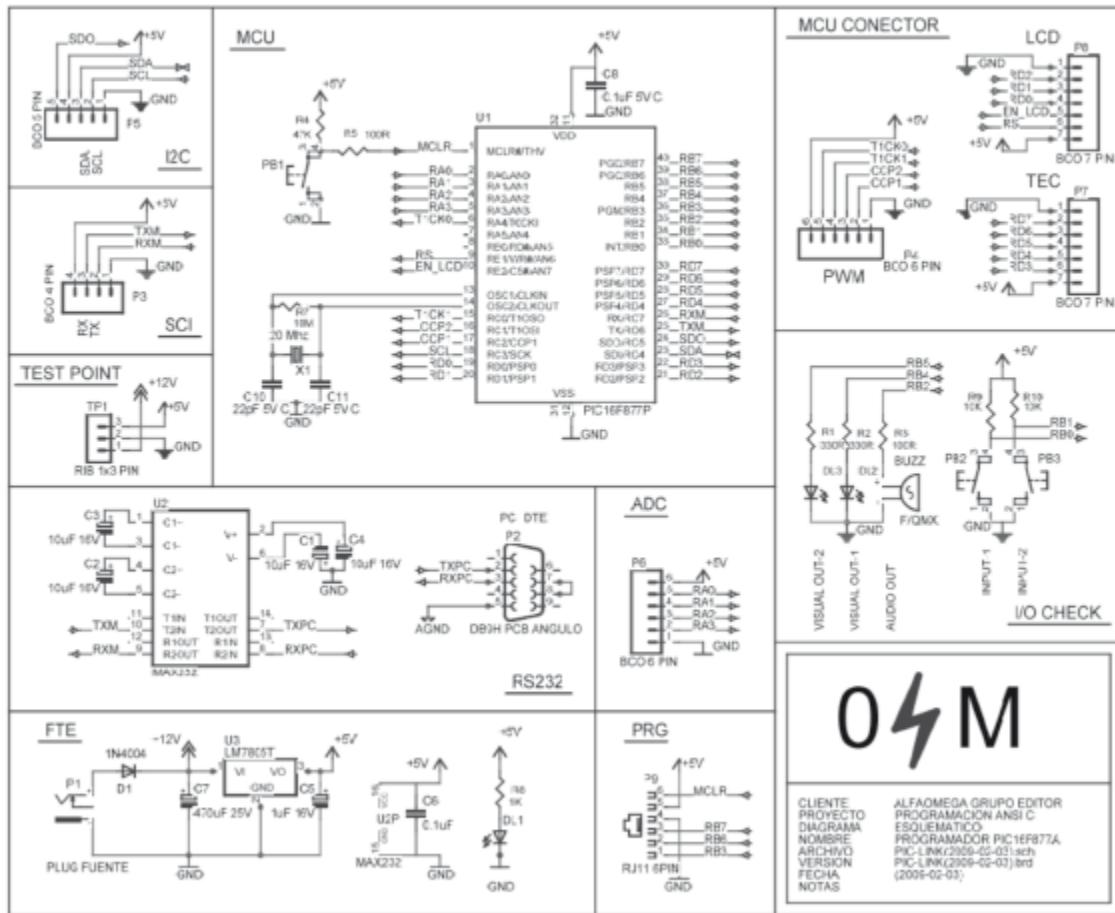


Pic-Link Microchip 16F8XX

Esta herramienta fue diseñada para usarse como interfaz al PC con el uso del ICD-U40...

**GRÁFICO
1.30**

Diagrama esquemático del sistema de evaluación PIC.

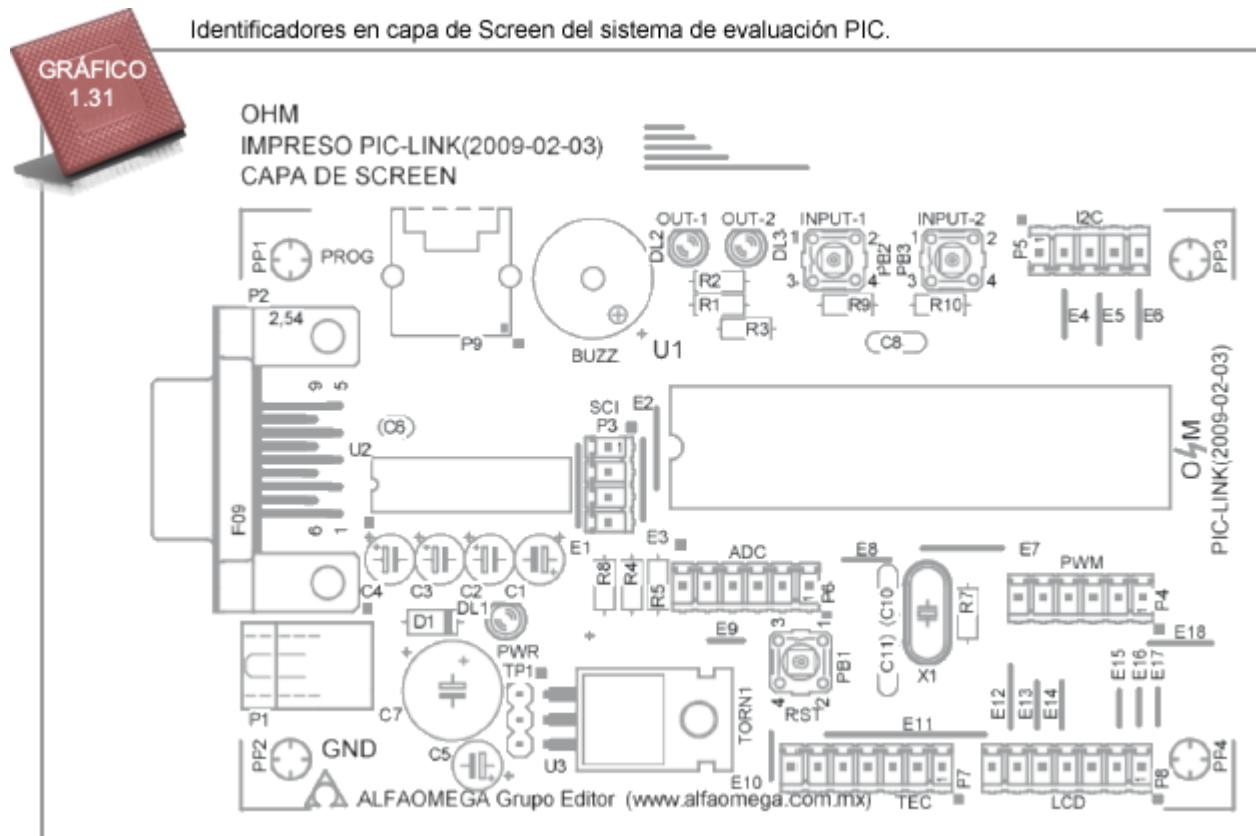


QTY	IDENTIFICADOR SCREEN PIC-Link	DESCRIPCIÓN Y VALOR	DIGIKEY P/N
1	BUZZ	Buzzer 5 Voltios Redondo	458-1061-ND
4	C1,C2,C3,C4	Condensador Aluminio 10uF 50V radial	P5567-ND
1	C5	Condensador Aluminio 1uF 50V radial	P5563-ND
2	C6,C13	Condensador Cerámico monolítico	399-4329-ND

		0.1uF 50V radial	
1	C7	Condensador Aluminio 470uF 25V radial	P5574-ND
2	C10,C11	Condensador Cerámico 22pF 50V	490-3708-ND
1	D1	Diodo 1N4004	1N4004DICT-ND
2	DL1,DL2	Diodo LED 3MM Rojo	516-1291-ND
1	DL3	Diodo LED 3MM Verde	516-1293-ND
18	E1,E2,E5,E6,E8,E9,E10, E11,E12,E13,E14,E15,E16,E17,E18	Puente Corto Circuito	NA
1	FTE	Adaptador de voltaje AC/DC 9VDC 0.66	T980-P5P-ND
1	GND	Punta de prueba negro	5001K-ND
1	P1	Plug Fuente para PCB pin interno pequeño	CP-202A-ND
1	P2	Conector DB9 para PCB Ángulo de 90 grados	A35107-ND
1	P3	Conector Blanco 4 Pines	WM4202-ND
2	P4,P6	Conector Blanco 6 Pines	WM4204-ND
1	P5	Conector Blanco 5 Pines	WM4203-ND
2	P7,P8	Conector 7 pines macho	WM2705-ND
3	PB1,PB2,PB3	Pulsador 4 pines para PCB 6mm	P12192S-ND
4	PP1,PP2,PP3,PP4	Soportes plásticos	NA
2	R6,R11	Resistencia 10K Ohm @ 1/4W	10KQBK-ND
1	R2	Resistencia 100 Ohm @1/4W	100QBK-ND
1	R7	Resistencia 10M Ohm @ 1/4W	10MQBK-ND
1	R8	Resistencia 1K Ohm @ 1/4W	1.0KQBK-ND
2	R10,R13	Resistencia 3.3K Ohm @ 1/4W	3.3KQBK-ND
1	R9	Resistencia 330K Ohm @ 1/4W(para X1=20MHz R9 debe ser un corto)	330KQBK-ND
1	R4	Resistencia 47K Ohm @ 1/4W	47KQBK-ND
6	TP1	Puntas de prueba para PCB	5000K-ND

1	TORN1	Tornillo 1/8" , 8 mm de largo con arandela y tuerca	NA
1	U1	Base de 40 pines TH, DIP 0.600	ED22426-ND
1	U1	Microcontrolador PIC16F877A	PIC16F877A-I/PND
1	U2	Integrado MAX232 16 pines DIP, R2232 Transceiver	MAX232ACPE-ND
1	U4	Regulador de Voltaje LM7805T TO-220	LM7805ECT-ND
1	X1	Oscilador 2 pines 20 MHz	CTX062-ND
1	P9	Conecotor telefónico	NA

Capa de Screen:



Capa de Componentes:

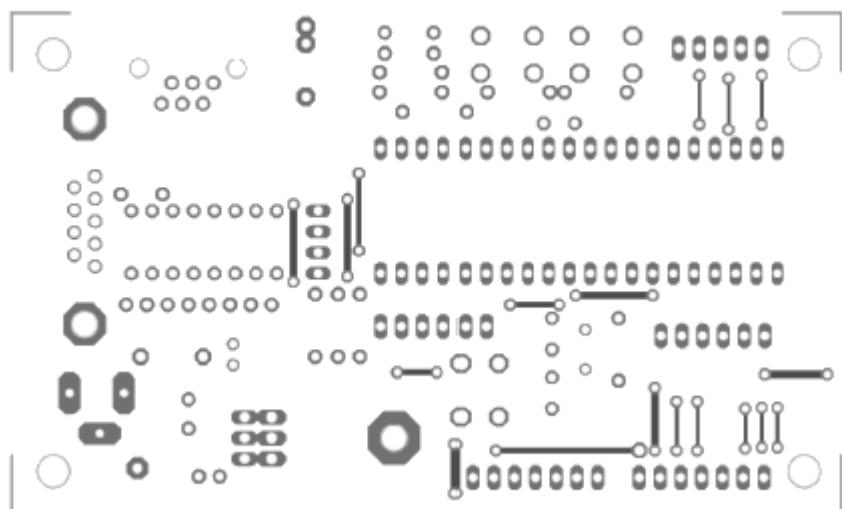


Posición de los puentes de corto circuito.

OHM

IMPRESO PIC-LINK(2009-02-03)

CAPA DE COMPONENTES



Capa de soldaduras:



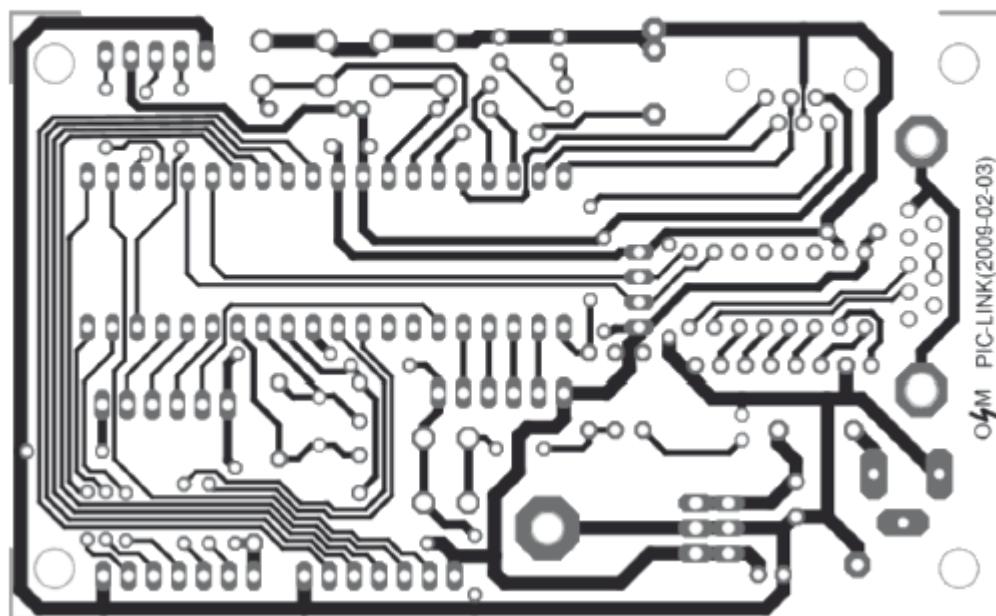
Conexiones en capa de soldadura sistema de evaluación PIC.

GRÁFICO
1.33

OHM

IMPRESO PIC-LINK(2008-05-03)

CAPA DE SOLDADURA



RESUMEN DEL CAPÍTULO

Un **compilador** es un programa que convierte un grupo de archivos texto en código de máquina de un procesador, a diferencia del **interpretador** cuyo código origen se envía a la máquina final (*target*) y va convirtiéndose en código de máquina en la medida en que se va ejecutando.

La programación en este texto está orientada a la compilación del código fuente por lo que gran parte del tiempo invertido en conversión de tiempo de solución de expresiones será resuelto en tiempo de compilación. El compilador tiene varias etapas para generar el código final, solucionando paso a paso todas las equivalencias, expresiones y análisis semántico, generando código intermedio que luego se unirá a las librerías que el programador invoca.

Las interrupciones en sistemas microcontrolados son eventos que suceden e interrumpen la máquina (CPU) en cualquier instante. Por esta razón, el código deberá ser diseñado de forma robusta, para que estas excepciones se manejen de forma adecuada y sin proveer errores al código elaborado, el procesador por su lado manejará sus prioridades, vectores y cambio de contexto, pero es el programador quien tendrá la responsabilidad final de su correcto funcionamiento.

La depuración en bajo nivel ayudará a identificar algunas causas por las cuales un programa en alto nivel no funcione adecuadamente, visualizando de cerca el comportamiento del PC (Contador de Programa), el SP (Apuntador al stack), variables en RAM y demás registros internos de la máquina.

Los buenos programadores en C para sistemas embebidos conocen muy bien la máquina y su arquitectura interna para la cual el compilador generará el código, tienen muy buen criterio y resuelven siempre los problemas por la vía más rápida y eficiente, sea por el lado del alto nivel C, o bien accediendo directamente a la máquina, sus pines, sus periféricos usando lenguaje ensamblador.

Existen en el mercado gran cantidad de herramientas disponibles, tanto de hardware como de software, que permiten al programador en alto y bajo nivel elegir lo que más se acomode a su presupuesto y requerimiento particular en tecnología, marca y desempeño.

Se recomienda elegir alguna que pueda cubrir varias familias de procesadores, con el fin tener flexibilidad a la hora de diseñar, de migrar o de expandir un sistema original. Es de gran utilidad una herramienta que permita realizar depuración en tiempo real (aunque sea con limitaciones), la cual permitirá la conexión entre el software desarrollado y la ejecución del mismo en el hardware embebido.

La fabricación de una herramienta propia con poca inversión “AP-Link” o “Pic-Link” permitirá ejecutar el código escrito en C y realizar depuración en tiempo real. Así como la ejecución real de todos los ejemplos desarrollados en el texto de forma práctica, conectando al programador con el mundo real, fortaleciéndolo en la ubicación de soluciones a los problemas que pueda encontrar en el desarrollo de sus proyectos.

El ensayo y error de los ejemplos, le convertirá en programador en C de alto desempeño, de muy buen conocimiento y manejo de una máquina particular (AP16A o PIC 16F877A) que será el camino para poder enfrentar a cualquier otro procesador que decida usar en los proyectos futuros: “*en realidad no importa en que marca o modelo de vehículo se aprenda a conducir, pero tendrá que ser en alguno en particular. Una vez que se aprende en éste, se podrá manejar el carro o camión que quiera*”

PREGUNTAS DEL CAPÍTULO

Mencione al menos 5 ejemplos de sistemas embebidos de la vida cotidiana.

¿Qué limitación tiene el hecho de usar interpretadores en lugar de compiladores en sistemas embebidos?

¿Puede el código objeto (.OBJ) compartirse con usuarios de procesadores de diferentes familias dentro del mismo fabricante o, entre procesadores de diferente marca?

¿Cómo se puede determinar las funciones que contiene una librería (.LIB), con el ánimo de usar alguna función?

Al adicionar comentarios a un código fuente (.C), ¿se alterará el tiempo de ejecución del target?

¿del PC que compila la aplicación final?

¿Bajo qué circunstancias de un proyecto seleccionaría un simulador de software o un emulador, como herramienta de desarrollo?

¿Será posible usar el COP del sistema microcontrolado como base de tiempo real, pensando que puede generar resets periódicos?

Estando el procesador dentro de una ISR, ¿Qué sucede si una interrupción de mayor prioridad solicita atención de la CPU? Realice una pequeña gráfica que ilustre la forma como el procesador realiza las tareas principales y las de ISR.

¿Qué ventaja tiene el esquema de interrupciones anidadas sobre el esquema no anidado?

INTRODUCCIÓN

El lenguaje ANSI C permite la programación de cualquier marca de microcontrolador, esto ayuda al diseñador del sistema embebido a no depender de una arquitectura o fabricante particular; sin embargo, es indispensable un conocimiento básico de la máquina específica que se programa, debido a varias razones:

- Porque es posible, de ser necesario, insertar en el proyecto algún código en lenguaje de ensamblador, que realice interfaz con el proyecto desarrollado en C (en este caso acceder a registros internos del procesador particular), e interactuar con el mapa de memoria.
- Existen situaciones en las cuales es necesario hacer depuración paso a paso a nivel de lenguaje máquina para encontrar algún “bug” que desde el alto nivel no es claro.
- Para tener criterio a la hora de decidir si el código que genera el compilador es el esperado en determinadas secciones de programa.
- Para tomar la decisión de realizar ciertas secciones en lenguaje de ensamblador.

Este capítulo presenta de forma muy resumida la arquitectura básica general de dos de las marcas más populares, conocidas y de fácil consecución en el mercado, como son la marca Microchip™ Technology Inc, y la marca Freescale™ Semiconductors.

Para la primera se examinará uno de los componentes de arquitectura RISC de 8 bits de mediano desempeño, referencia PIC16F877A, y para la segunda marca uno de los miembros de la familia de arquitectura CISC de 8 bits HC08 referencia MC68HC908AP16A.

Con base en estos dos componentes se ilustrará de forma general la estructura interna de cada marca, pasando por sus registros internos, distribución de pines, mapas de memoria y modos de direccionamiento.

Es importante conocer los detalles más significativos, porque a lo largo del texto estos dos procesadores serán los usados para las prácticas específicas.

El capítulo está orientado a los programadores de alto nivel que requieren programar un microcontrolador en un sistema embebido, más que a los programadores de bajo nivel de lenguaje ensamblador; estos últimos conocen muy bien sus máquinas, sin embargo, se recomienda la lectura en caso que en el pasado no trabajara con alguna de estas dos arquitecturas.

Se termina con un ejemplo práctico realizado puramente en lenguaje de ensamblador.

2.1 Arquitectura RISC (Harvard) Microchip de 8 bits

RISC: *Reduced Instruction Set Computer*, esta arquitectura contiene buses separados para la decodificación de instrucciones y datos, y es la denominada arquitectura Harvard.

Este doble bus permite que la gran mayoría de las instrucciones sean ejecutadas en un solo ciclo de máquina.

De forma general, la familia de microcontroladores de



Los
microcontroladores
se clasifican
fundamentalmente

Microchip™ están clasificados en 3 grupos:

Core de 12 bits línea básica, series 10,12 y 16.

Core de 14 bits de medio rango, series 12 y 16.

Core de 16 bits, serie 18.

por el número de bits que contienen en el ancho del bus de datos.



Para cada uno de los grupos aplican varias reglas, pero la clasificación más amplia la realiza el número de bits en la palabra de la instrucción que corresponde al ancho de la palabra; sin embargo, todos están clasificados dentro del grupo de procesadores de 8 bits, que corresponde al ancho del bus de datos.

El caso más popular de la línea de productos de 8 bits son los de medio rango que contienen core de 14 bits y es acá donde el componente particular PIC16F877A está clasificado.

La serie de core de 12 bits es menos sofisticada que la línea de core de 14 bits y solo es recomendada para aplicaciones de muy bajo costo y que no requieran mucho desempeño.

Recientemente Microchip™ ha introducido al mercado su nueva familia de microcontroladores de 16 bits, donde tanto el bus de datos como el de direcciones es de 16 bits de longitud, contienen mejor desempeño especialmente al ser trabajados en lenguaje de alto nivel como el C, de tal forma que las limitaciones que se han mencionado del C cada vez tienen menos importancia.



Microchip™ ha desarrollado una nueva familia de microcontroladores de 16 bits, donde tanto el bus de datos como el de direcciones es de 16 bits de longitud, esto le otorga un mejor desempeño al ser trabajados en lenguaje de alto nivel como el C.

El procesador puede direccionar los datos de forma directa o indirecta. En los microprocesadores de arquitectura Harvard

Algunas de las características de la CPU RISC de Microchip son las siguientes:

Velocidad de operación hasta 20 MHz.

Capacidad de manejo de interrupciones.

Profundidad de stack de 8 niveles.

Modos de direccionamiento directo, indirecto y relativo.

35 instrucciones, todas de un ciclo excepto los saltos.

Los procesadores Microchip™ de alto desempeño tienen gran cantidad de características comunes de la arquitectura de procesadores RISC. Su arquitectura Harvard permite que una instrucción sea ejecutada, mientras la siguiente está siendo traída de la memoria de programa (*fetched*).

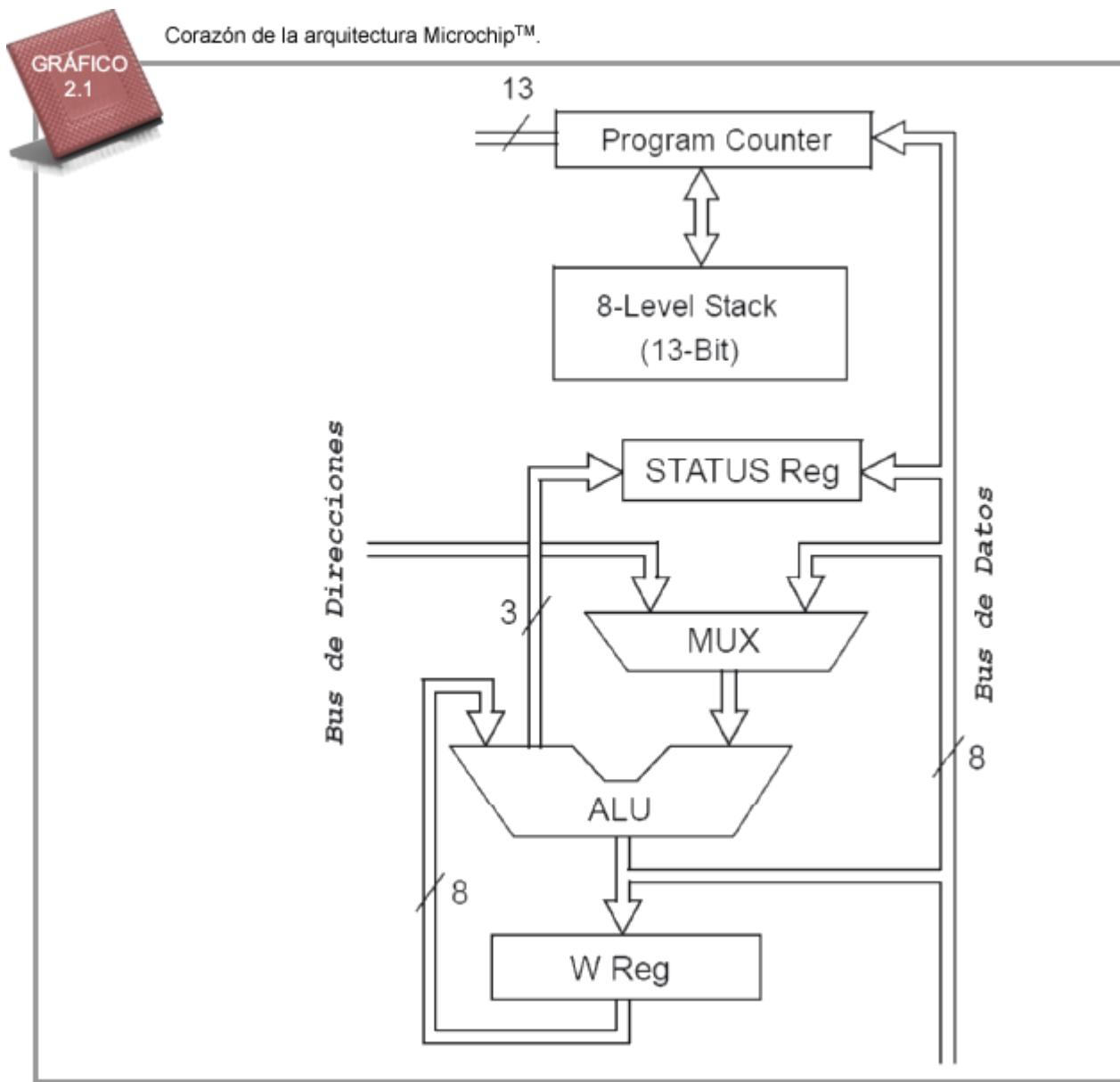
Esta familia de procesadores puede de forma directa o indirecta direccionar sus registros o memoria de datos. Existe a su vez una

cuando una instrucción está siendo ejecutada, la siguiente es traída a la memoria de programa con lo cual se mejora su desempeño.

zona donde se encuentran los registros especiales denominados SFR (*Special Function Registers*), los cuales incluyen los registros del core, como son el contador de programa (PC), el registro de estado (SR) y los registros de control de los periféricos internos que están mapeados en la memoria de datos.

Una de las características que también lo hacen eficiente es la ortogonalidad en las instrucciones, que hace posible ejecutar cualquier operación, en cualquier registro, usando cualquier modo de direccionamiento. Esta simetría hace que la programación sea eficiente y la curva de aprendizaje reducida, porque no es necesario el aprendizaje particular de cada uno de sus modos de direccionamiento e instrucciones.

2.1.1 Componentes básicos de la arquitectura Microchip™



Unidad aritmética lógica (ALU)

ALU: *Aritmetical Logic Unit*, es el corazón del procesamiento del microcontrolador Microchip™, lo constituye una unidad de 8 bits de propósito general. Realiza funciones aritméticas y booleanas entre los datos de cualquier registro en el mapa de memoria del procesador.

Muchas de las operaciones se hacen con el registro de trabajo (W), con los registros de propósitos generales, alterando con cada resultado el registro de estado (SR).

Registro de trabajo (W Reg)

El **W** es un registro de 8 bits, usado para operaciones y resultados de la ALU, aunque tiene localización física en el mapa de memoria, no puede ser direccionado por ninguna instrucción. Además es usado para almacenar argumentos de una función, y también como registro de almacenamiento temporal de resultados.

Contador de programa (PC)

El contador de programa es un registro interno de 13 bits, que contiene la dirección de la instrucción que está siendo ejecutada por el procesador. Equivale a uno de los SFR del mapa de memoria de datos del PIC.

Su valor está contenido en 2 direcciones la 0x02 que contiene la parte baja y la 0x0A que contiene 5 bits de la parte alta.

02h	PCL	Program Counter (PC) Least Significant Byte				
0Ah	PCLATH	—	—	—	—	Write Buffer for upper 5 bits of Program Counter

Registro de estado (Status Reg → SR)

Este registro contiene el estado de la ALU, el bit de estado de *Reset* y los bits de selección de banco de la memoria RAM.

Dependiendo de la instrucción ejecutada, la ALU puede afectar los valores de cada uno de los bits que se describen a continuación:

03h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
-----	--------	-----	-----	-----	----	----	---	----	---

IRP → Bit de selección de banco, usado para el direccionamiento indirecto.

1: Banco 2, 3 (0x0100 a 0x01FF).

0: Banco 0, 1 (0x0000 a 0x00FF).

RP1: RP0 →Registros de selección de banco, usado para direccionamiento directo.

00: Banco 0 (0x000 a 0x07F).

01: Banco 1 (0x080 a 0x0FF).

10: Banco 2 (0x100 a 0x17F).

11: Banco 3 (0x180 a 0x1FF).

TO →Bit de Time Out

1: Despues del arranque, instrucciones CLRWD or SLEEP.

0: Ocurrió un time out del WDT (*watchdog timer*).

PD → Bit de Power-down

1: Después del arranque o por efecto de la instrucción CLRWDT.

0: Por ejecución de la instrucción SLEEP.

Z →Zero (Cero)

1: El resultado de la operación aritmética o lógica es cero.

0: El resultado de la operación aritmética o lógica es diferente de cero.

DC → Digit Carry

1: La anterior instrucción generó un acarreo del cuarto bit.

0: No se generó acarreo en la última instrucción ejecutada.

C →Carry (acarreo).

1: Ocurrió un acarreo en el bit de mayor peso.

0: No se generó acarreo en el bit de mayor peso.

Stack y apuntador de Pila

El *stack* es una zona que contiene espacio de 8 niveles de 13 bits de ancho, usado para guardar temporalmente la dirección del PC (Contador de Programa) cuando una instrucción de llamado de subrutina es invocada (instrucción CALL), o una llamada automática de una interrupción es generada.

El *stack pointer* es un registro interno que contiene la dirección actual del nivel dentro de esta zona de *stack*.

Los valores que contiene el *stack* son recuperados una vez que la instrucción de retorno de subrutina (instrucción *return*), o de retorno de interrupción (instrucciones RETLW o RETFIE), funciona como una lista LIFO (*Last Input First Output*).

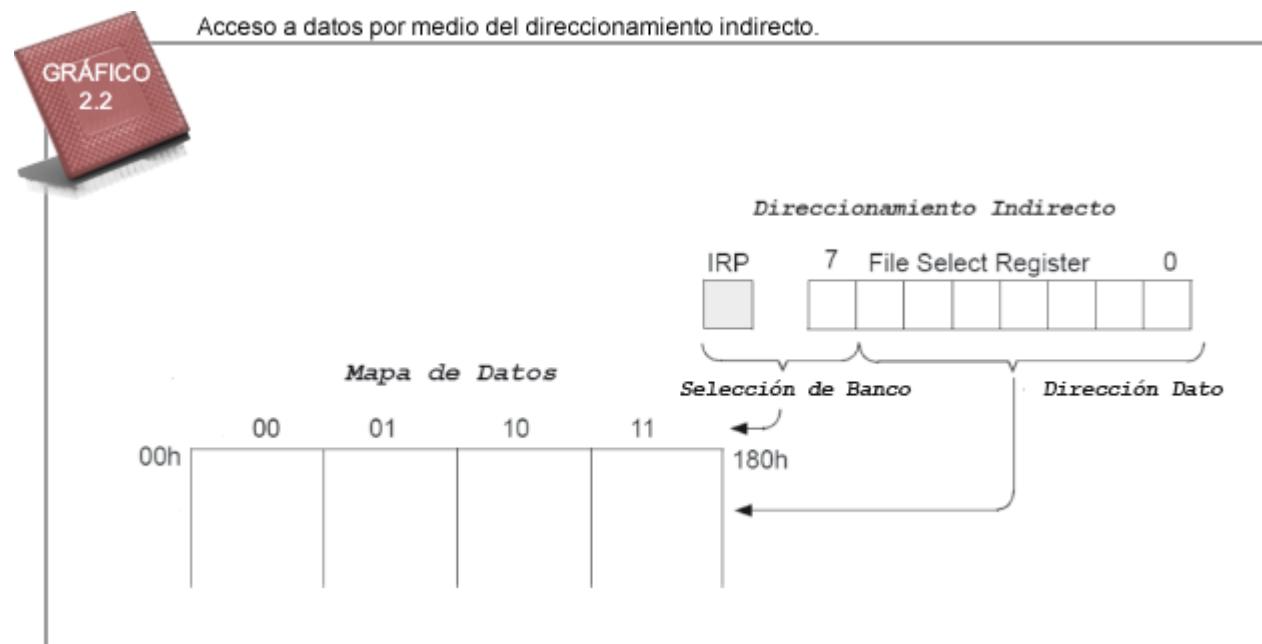
El registro PCLATH no es afectado por estos almacenamientos y recuperación hacia y desde el *stack*.

El *stack* opera como un buffer circular, esto significa que después que se han ingresado ocho (8) valores, el noveno (9) valor sobrescribe sobre el primer valor, y el décimo sobre el segundo valor y así sucesivamente; es por esta razón que el programador debe administrar de forma responsable su uso.

2.1.2 Modos de direccionamiento Microchip

Direccionamiento indirecto

El registro INDF no es registro físico dentro del mapa de memoria, su acceso causará un direccionamiento indirecto.



Cualquier instrucción que use el registro INDF accesará el dato apuntado por el FSR (*File Select Register*). Una dirección efectiva de 9 bits es obtenida por la concatenación del FSR (8 bits) y el bit IRP (SR<7>).

00h	INDF	Adressing this location uses contents of FSR to address data memory (not a physical register)
04h	FSR	Indirect Data Memory Address Pointer

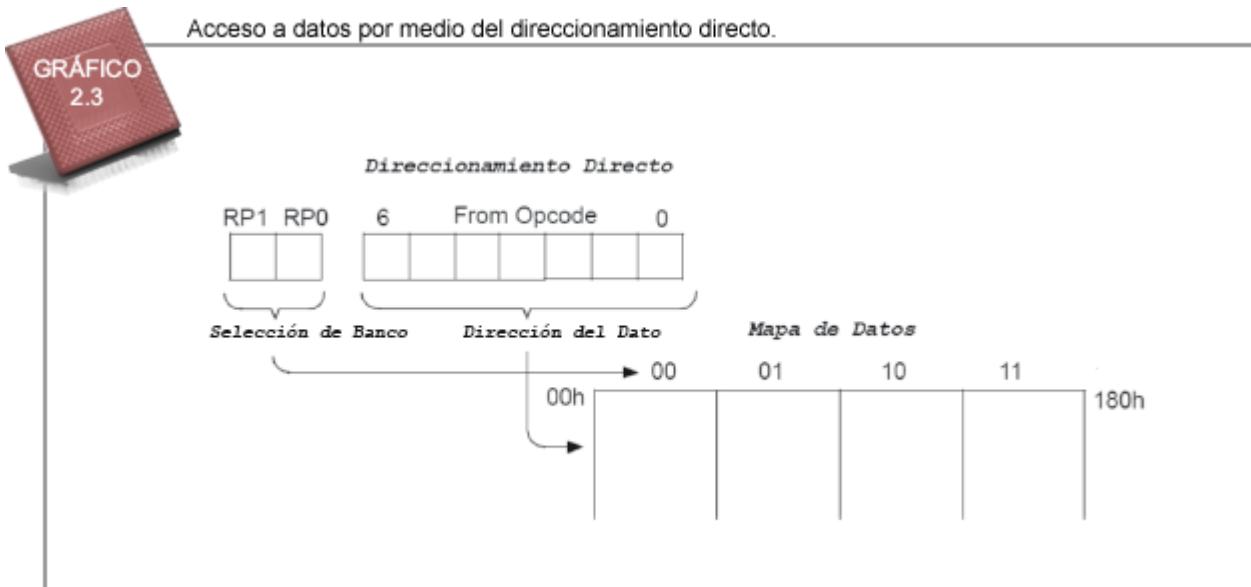
El siguiente código muestra cómo realizar acceso usando este modo de direccionamiento, esta sección de programa realiza el borrado de la localización de RAM desde la dirección 0x0020 a 0x002F.

```
MOVLW 0x20      ;Inicializa el Apuntador
MOVWF FSR       ;a RAM
NEXT  CLRFL INDF  ;Inicializa el Registro INDF
      INCF FSR    ;Incrementa el apuntador
      BTFSS FSR,4  ; terminó?
      GOTO NEXT   ;no, regrese de nuevo a NEXT
                  ;sí, continúe
```

Direccionamiento directo

En este tipo de direccionamiento, el valor con el que operará la máquina está contenido en el opcode.

Dependiendo del tipo de instrucción será este el dato sobre el cual se realice el ciclo de lectura o de escritura.



El siguiente código mueve el dato contenido en la dirección 0x20 a la 0x40.

```
MOVF 0x20,W      ;muestra el valor que contiene 0x20 al Reg W.  
MOVWF 0x40        ;almacena el contenido de W en la dirección 0x40
```

Direccionamiento relativo

El direccionamiento relativo es usado en instrucciones que toman decisión al final de ciclo de instrucción para realizar o no una alteración del flujo normal del contador de programa ($PC = PC + 1$).

La decisión de alterar el PC depende del tipo de instrucción y el resultado final de una operación sobre un dato o un bit del dato.

Este tipo de instrucciones se tardan un ciclo más en ejecutarse, debido a que no es posible hacer el procesamiento de la siguiente instrucción hasta no conocer antes el nuevo valor del PC.



Direccionamiento relativo Microchip™.

El siguiente código realiza un pequeño retardo, permaneciendo en el loop 10 iteraciones, en la instrucción DECFSZ toma de decisión de salto si el valor del registro cnt llegó a 0 (cero).

```
MOVLW 0x0A      ;carga el Reg W con el valor decimal  
MOVWF cnt       ;almacena el contenido de W en la variable cnt  
LOOP  DECFSZ cnt,1 ; Decrementa y examina si llegó a cero?  
GOTO   LOOP       ;sino salta de nuevo a LOOP
```

2.1.3 Mapa de memoria Microchip del PIC16F877A

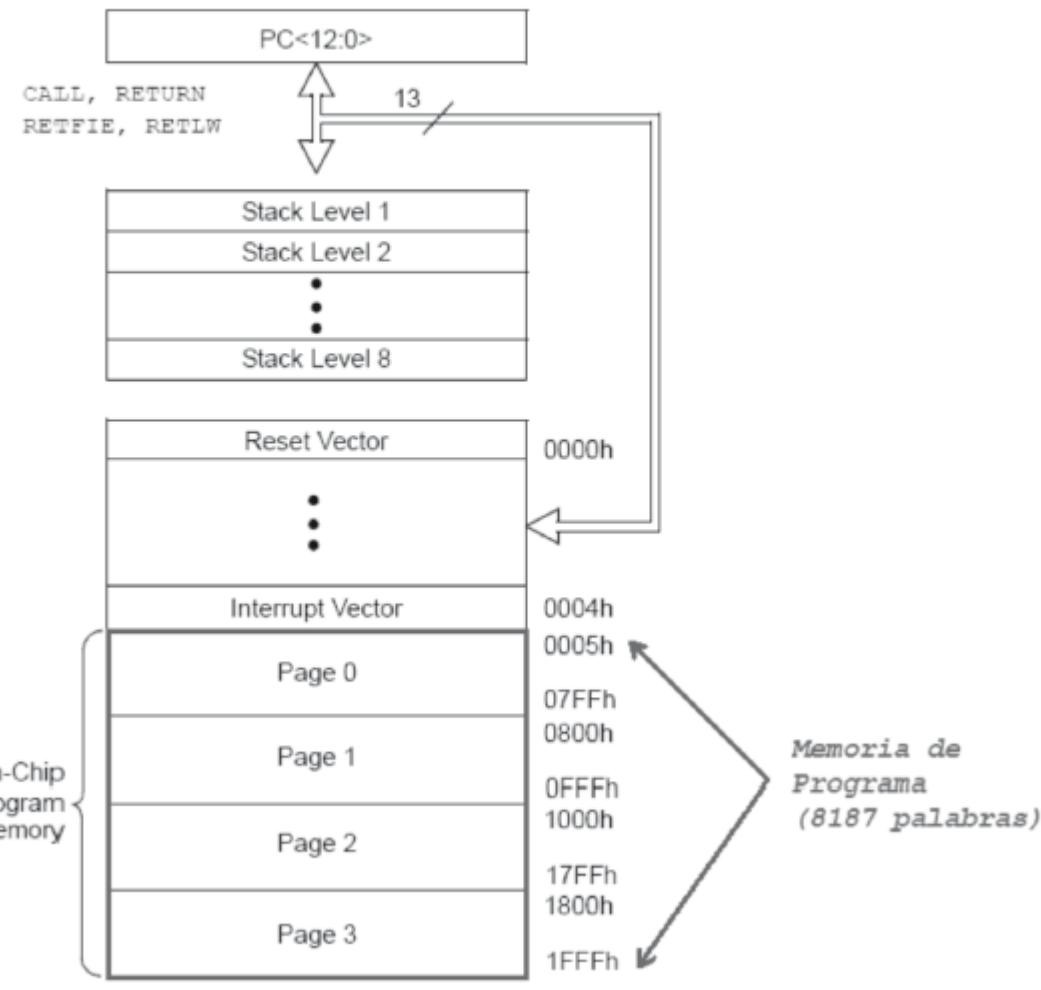
Mapa de memoria de programa

Esta familia de microcontroladores contiene un contador de programa de 13 bits que puede direccionar un espacio de memoria de 8K x palabras de 14 bits.

Para el caso particular del PIC16F877A el direccionamiento va desde la dirección más baja 0x0000 hasta la dirección 0x1FFF (8K de memoria), donde el vector de *Reset* ocupa la dirección 0x0000 y el vector de interrupción está en la dirección 0x0004, de tal forma que el espacio disponible para el código del usuario se encuentra desde la dirección 0x0005 hasta la dirección 0x1FFF (8187 palabras).

**GRÁFICO
2.5**

Mapa de memoria de programa del microcontrolador PIC16F877A.



Mapa de memoria de datos

Para accesar a los bancos de memoria de datos se modifican los bits RP1 Y RP0.

La memoria de datos está posicionada en 4 bancos que contienen los registros de propósito general (GPRs) y los registros de funciones especiales (SFR), los cuales están localizados en las primeras 32 localizaciones de cada banco (*ver Gráfico 2.6*).

Las demás localidades del mapa de datos están designadas para los GPRs que se ilustran en el *Gráfico 2.7*.

Las direcciones **0xF0** a **0xFF** del banco 1, **0x170** a **0x17F** del banco 2, **0x1F0** a **0x1FF** del banco 3, están implementadas como área común de RAM y corresponden a las mismas direcciones **0x70** a **0x7F** del banco 0.

El acceso a cada uno de los bancos se realiza mediante la modificación de los bits RP1 y RP0 en el registro SR como lo ilustra el *Gráfico 2.6*.

GRÁFICO
2.6

Acceso a los bancos de RAM del microcontrolador PIC16F877A.

Bank	RP1	RP0
0	0	0
1	0	1
2	1	0
3	1	1

**GRÁFICO
2.7**

GPRs: Registros de propósito general del microcontrolador PIC 16F877A.

Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD ⁽²⁾	08h	TRISD ⁽²⁾	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2 ⁽¹⁾	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADDRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers	116h	General Purpose Registers	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h	16 Bytes	119h	16 Bytes	199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers		20h	A0h	General Purpose Registers	120h	General Purpose Registers	1A0h
96 Bytes		3Fh		80 Bytes		80 Bytes	
		40h		80 Bytes		80 Bytes	
		6Fh		80 Bytes		80 Bytes	
		70h		accesses 70h-7Fh		accesses 70h-7Fh	
		7Fh		accesses 70h-7Fh		accesses 70h-7Fh	
Bank 0		Bank 1		Bank 2		Bank 3	

2.1.4 Características del microcontrolador PIC16F877A

El PIC16F877A es uno de los miembros de la familia de los microcontroladores de bajo costo y alto desempeño de 8 bits con las siguientes características básicas:

Opciones de oscilador externo o interno.

Modos de bajo consumo SLEEP.

Módulo Watchdog de bajo consumo.

33 entradas/salidas configurables.

2 comparadores analógicos.

8 canales de ADC de 10 bits de resolución.

Watchdog timer con oscilador Independiente.

Programación en circuito.

Circuito de encendido (Brown-Out y Power-On Reset y Power-Up). Voltaje de operación de 2.0V a 5.5V.

Capacidades de memoria: flash de programa 8192 palabras, memoria RAM de 368 bytes y 256 bytes de E2prom.

3 módulos de temporización: Timer0 (8 bits), Timer1 (16 bits), Timer2 (8 bits).

USART/SCI de comunicaciones seriales.

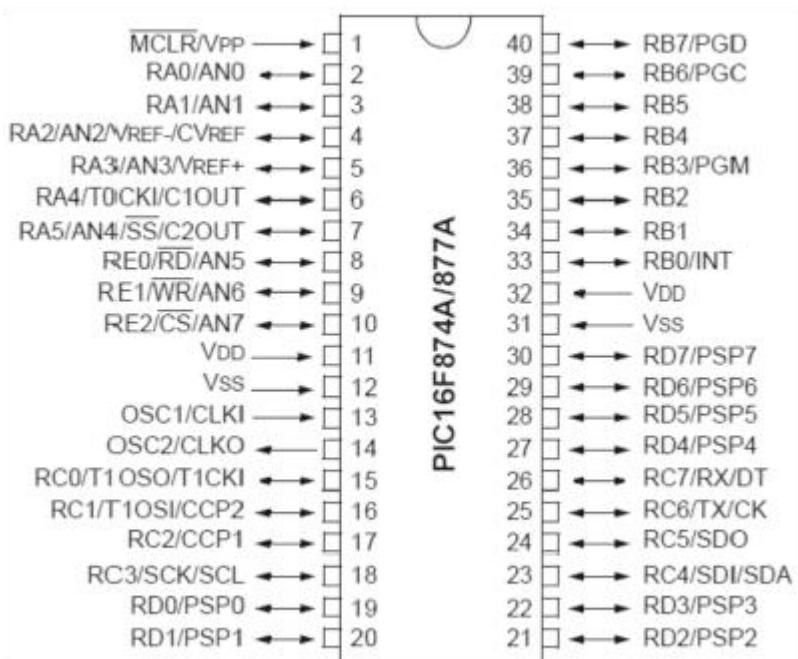
Memoria EEPROM de un millón de escrituras.

Módulo de entrada de captura, salida por comparación, y PWM.

En las siguientes dos figuras se ilustra su diagrama de pines y su diagrama de bloques interno.

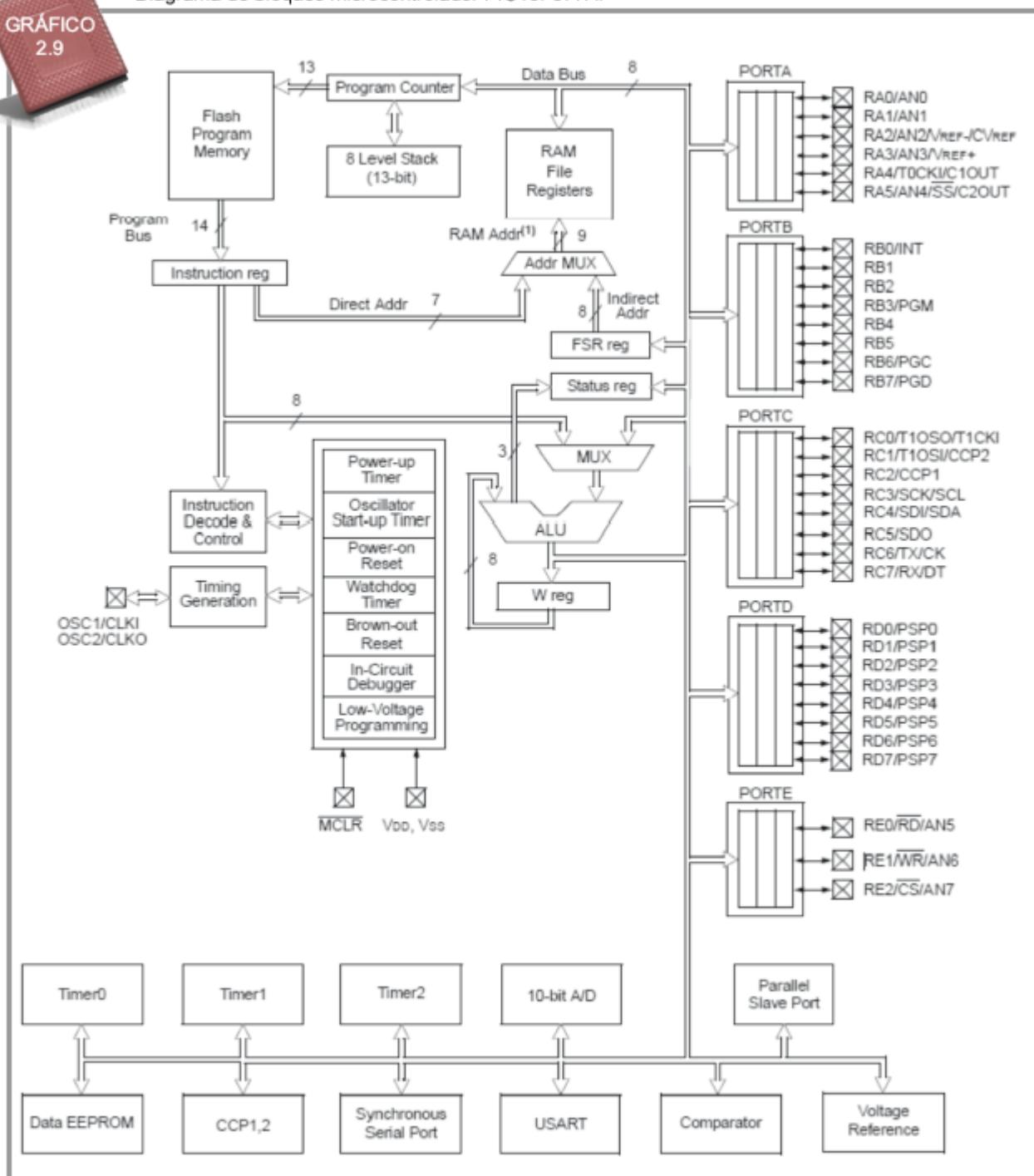
**GRÁFICO
2.8**

Diagrama de pines del microcontrolador PIC16F877A.



**GRÁFICO
2.9**

Diagrama de bloques microcontrolador PIC16F877A.



2.1.5 Fuentes de interrupción del microcontrolador P IC16F877A



El procesador Microchip PIC16F877A permite controlar las fuentes de interrupción de manera independiente, mejorando la administración de los procesos en el programa.



Las opciones INTCON, PIE1 y PIE2 son las encargadas de habilitar o deshabilitar las fuentes de interrupción en el procesador Microchip PIC16F877A.

(ver Gráfico 2.10, 2.11 y 2.12).

Dependiendo de los periféricos específicos de cada microcontrolador Microchip™, existen también fuentes de interrupción adicionales.

El procesador que PIC16F877A es uno de los más completos dentro de la línea de Microchip™ y ofrece gran cantidad de interrupciones, tanto de hardware como de software, que alivian la carga del procesador central, permiten administrar y controlar los diferentes procesos y de forma notable mejorar los niveles de consumo de energía.

Cada una de las diferentes fuentes de interrupción puede ser controlada de manera independiente, de tal forma que es posible solo tener algunas fuentes generando interrupciones, mientras que otras no generan ningún efecto en el desarrollo normal del programa principal que está ejecutando el procesador.

El procesador contiene 3 registros especiales llamados el **INTCON** (*Interrupt Control Register*), **PIE1** (*Peripheral Interrupt Enable Register 1*) y **PIE2** (*Peripheral Interrupt Enable Register 2*), de lectura y escritura que permiten con cada uno de sus bits habilitar o deshabilitar alguna fuente de interrupción particular

Sin embargo, para que sucedan las interrupciones que están habilitadas, deberán encenderse los bits **GIE** (*Global Interrupt Enable*) y **PEIE** (*Peripheral Interrupt Enable*) del registro **INTCON** (ver Gráfico 2.10).



Registro INTCON.

GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
-----	------	------	------	------	------	------	------



Registro PIE1 de habilitación de interrupciones de periféricos.

-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
---	------	------	------	-------	--------	--------	--------



Registro PIE2 de habilitación de interrupción de periféricos.

OSFIE	C2IE	C1IE	EEIE	BCLIE	ULPWUIE	-	CCP2IE
-------	------	------	------	-------	---------	---	--------

Las fuentes de interrupción de este procesador son:

Sobreflujo del Timer0

El Timer0 genera una interrupción cuando el registro **TMR0** (contador del timer 0), llega al valor de 0xFF y pasa a 0x00, siempre y cuando el bit **TOIE** (*Timer 0 Overflow Interrupt Enable*) se encuentre en 1.

El bit de **TOIF** (*Timer Overflow Interrupt Flag*) en el registro **INTCON** se encenderá indicando la ocurrencia del evento y esta misma bandera deberá ser borrada antes de abandonar la función de interrupción ISR.

Cambio en los pines del puerto B

Se presenta una interrupción cuando el estado digital de alguno de los pines del Puerto B (RB0 – RB7) del microcontrolador cambie de estado.

La bandera RBIF (**PORTB change Interrupt Flag**) se encenderá indicando el suceso, y deberá ser borrada por software, lo que indica que la atención a la interrupción fue reconocida.

Para habilitar que el suceso de esta interrupción el bit RBIE (*PORTB change Interrupt Enable*), deberá estar en 1.

Interrupción externa

La unidad generará una interrupción si el pin RB0 del Puerto B cambia de estado, el flanco, (bajada o subida) en el cual la interrupción es generada, se controla con el bit INTEDG en el registro OPTION_REG.

El suceso de esta interrupción es controlada mediante el bit INTE (*INT External Interrupt Enable bit*) en el registro **INTCON**.

Fin de conversión del ADC

Esta fuente genera interrupción una vez que el módulo Conversor ADC (Analógico a Digital), ha terminado la conversión, y el valor resultante puede ser leído de su registro.

El control de esta interrupción lo realiza el bit **ADIE** en el registro **PIE1**.

Transmisor del SCI vacío

Genera interrupción una vez que el registro trasmisor de datos seriales SCI está vacío, indicando que el dato del periférico fue enviado al exterior y está listo para recibir un nuevo dato de la trama.

El control de esta interrupción se realiza modificando el bit **TXIE** en el registro **PIE1**.

Dato disponible en el receptor del SCI

Genera una interrupción cuando el registro de receptor de datos seriales del SCI contiene un dato nuevo recibido del exterior.

El control de esta interrupción se realiza mediante el bit **RCIE** en el registro **PIE1**.

Sobreflujo del Timer 1

Genera una interrupción cuando el Timer 1 de 16 bits del microcontrolador sobrepasó su nivel máximo (0xFFFF) y regresó a 0 (0x0000).

El control de la interrupción se realiza mediante el bit **TMR1IE** en el registro **PIE1**.

Sobreflujo del Timer 2

Genera una interrupción cuando el Timer 2 de 8 bits del microcontrolador sobrepasa su nivel máximo (establecido en el registro **PR2**) y regresa a 0 (0x00).

El control de la interrupción se realiza mediante el bit **TMR2IE** en el registro **PIE1**.

Captura o comparación en la Unidad 1

Genera interrupción una vez que se detecta un suceso de entrada de captura o salida por comparación en la Unidad 1.

El control de esta interrupción se realiza mediante el bit **CCP1IE** en el registro **PIE1**.

Captura o comparación en la Unidad 2

Genera interrupción una vez que se detecta un suceso de entrada de captura o salida por comparación en la Unidad 2.

El control de esta interrupción se realiza mediante el bit **CCP2IE** en el registro **PIE2**

Puerto serial sincrónico MSSP

Dependiendo del modo de funcionamiento (I2C o SPI) del módulo MSSP genera interrupción si una trasmisión o recepción se realizó de forma completa.

La ocurrencia o no de esta interrupción se controla mediante el bit **SSPIE** en el registro **PIE1**.

Colisión de bus

Esta fuente de interrupción solicita atención del procesador cada vez que una colisión en el bus I2C ha ocurrido, en el módulo MSSP cuando está configurado en modo maestro.

El control de ocurrencia de esta interrupción lo tiene el bit **BCLIE** en el registro **PIE2**.

Escritura en EEPROM completa

Se presenta una interrupción cada que una escritura a la memoria interna EEPROM sea finalizada; indica además, que el periférico está listo para recibir un nuevo byte de la trama de datos a ser programada.

El control se realiza mediante el bit **EEIE** en el registro **PIE2**.

Falla en el Sistema de oscilador

Genera interrupción cada vez que el procesador detecte que la fuente de oscilador presenta una falla, en este caso el procesador cambia su funcionamiento a la fuente interna **INTOSC**. El control de esta interrupción lo realiza el bit **OSFIE** en el registro **PIE2**.

Detección de Comparación 1

Generará interrupción el módulo de comparación de voltaje 1 cuando se presente una comparación exitosa. El control se realiza mediante el bit **C1IE** en el registro **PIE2**.

Detección de Comparación 2

Generará interrupción el módulo de comparación de voltaje 2 cuando se presente una comparación exitosa. El control se realiza mediante el bit **C1IE** en el registro **PIE2**.

2.2 ARQUITECTURA CISC (VON NEWMAN) FREECALE DE 8 BITS

La arquitectura CISC (*Complex Instruction Set Computer*) es una estructura interna de diseño de procesadores en la que el bus de datos está compartido con el bus de direcciones.

En el caso particular de los procesadores Freescale™ de 8 bits, el bus de direcciones es de 16 bits y el bus de datos es de 8 bits, por lo que solo es necesario compartir la mitad del bus.



Los procesadores basados en la arquitectura CISC tienen compartido el bus de datos con el de direcciones, lo que permite realizar operaciones aritméticas de mayor complejidad que los basados en la arquitectura RISC.



Esta arquitectura permite tener instrucciones más elaboradas que realizan operaciones aritméticas más complejas que la arquitectura **RISC**.

Sin embargo, por tener un solo bus compartido el tiempo de ejecución se puede ver un poco alterado; por su parte, existe un único mapa de datos continuo para los datos de programa (código del software) y para los datos de almacenamiento temporal RAM (datos del programa), que permite direccionar de igual forma y mediante las mismas instrucciones los datos de la memoria Flash y RAM, incluso permitir la ejecución de programa en la memoria de almacenamiento RAM.

Esta última es una ventaja de la arquitectura CISC sobre la RISC si es lo que el programador quiere; sin embargo, puede ser una desventaja si el procesador es llevado a la RAM por algún ruido o error en el código programado, el procesador puede ejecutar instrucciones no deseadas hasta generar un *Reset* no previsto.

2.2.1 Modelo de programación Freescale HC(S) 08

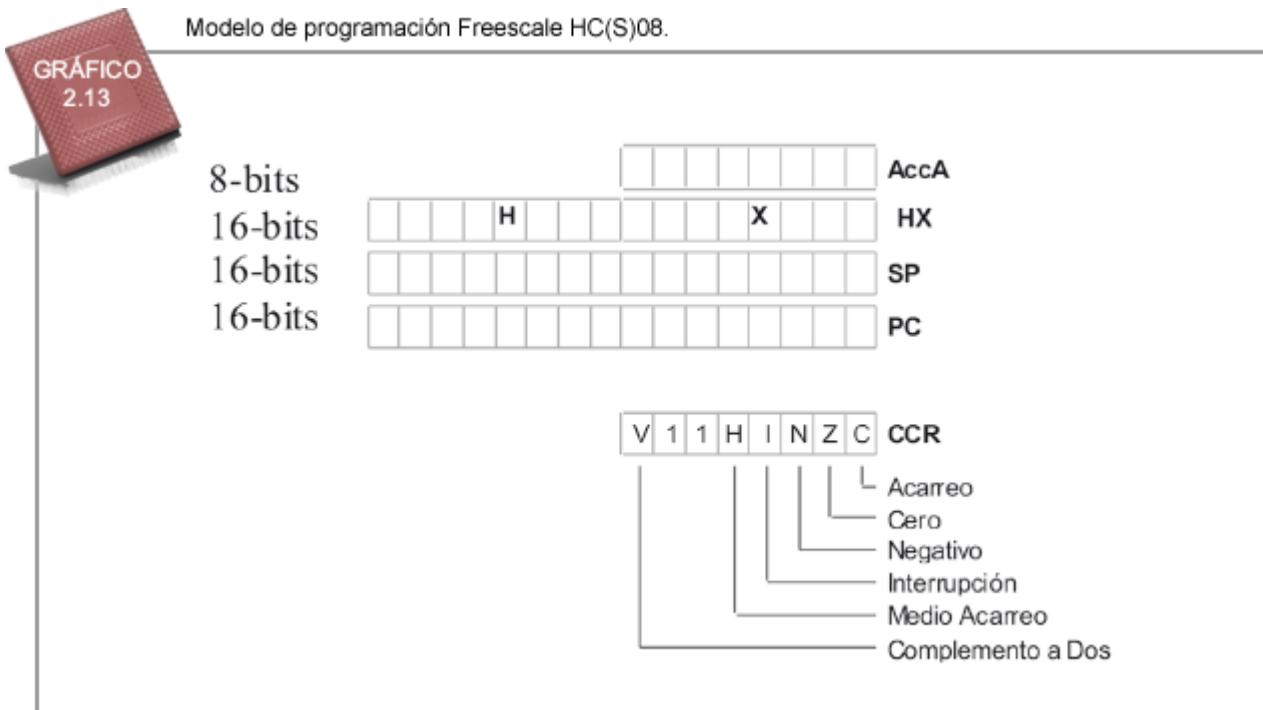
El modelo de programación es el corazón del procesamiento de la CPU 08; lo constituyen registros de trabajo en los que se almacenan resultados temporales, argumentos de funciones y estado actual de la máquina. Estos registros internos no aparecen dentro del mapa de memoria del procesador y son guardados de forma automática una vez una interrupción es llamada.



En los procesadores Freescale HC,

Todos los miembros de la familia HC08 y HCS08 tienen un mismo modelo de programación, las mismas instrucciones, esto indica que cualquier código compilado y optimizado será igual para cualquiera de los procesadores de 8 bits.

cuando se produce una interrupción, el modelo de programación es almacenado en el stack.



El modelo CPU 08 lo constituyen los siguientes registros:

Acumulador A (AccA)

Acumulador de 8 bits de propósito general, la CPU lo usa para almacenar operandos y resultados de algunas operaciones, normalmente de tipo aritmético o lógico. El compilador puede convertir algún código que usa el acumulador A para almacenar argumentos de 1 byte.

Registro índice (H: X)

Este registro está conformado por 2 partes, el byte alto llamado H y la parte baja llamada X.

El registró concatenado de 16 bits H: X, permite el direccionamiento indexado, el cual accesa todo el mapa de memoria de 64K. Por compatibilidad con la familia anterior HC05 el registro H es inicializado en cero al salir de *reset*. El compilador suele usarlo para manipular datos enteros (2 bytes), entregar argumentos a funciones y manejar tablas, arreglos y estructuras.

Apuntador de Pila (SP)

Es un registro de 16 bits que contiene la dirección de la siguiente dirección disponible en la pila (*stack*). Una vez se sale de reset el SP se inicializa en 0x00FF, esto para mantener compatibilidad con su familia predecesora, la HC05.

La dirección en el SP se decremente cada vez que algún dato es empujado o almacenado en el *stack* y se incrementa cada vez que datos son recuperados del *stack*.

El SP permite además, el manejo de instrucciones que usan su valor como dirección base de acceso en el direccionamiento orientado al *stack*.

Contador de programa (PC)

Es un registro de 16 bits que contiene la dirección de la siguiente instrucción a ser ejecutada. Normalmente su dirección es incrementada automáticamente a la siguiente dirección de memoria cada vez que una instrucción o dato es procesado; las instrucciones Jmp, branch, o interrupción posicionan el PC en un valor no secuencial.



Al momento de *reset* el valor del PC es el contenido en las direcciones 0xFFFFE y 0xFFFF donde es iniciado el programa, normalmente apuntan a la dirección de la rutina de inicio en C o “*Startup*”.

Registro de banderas (CCR)

CCR: *Condition Code Register*. Es un registro de 8 bits, que contiene el bit de enmascaramiento general de interrupciones y 5 banderas que indican resultados de la instrucción previamente ejecutada.

V → Bandera de rebosamiento: Se pone en “1” cuando ocurre un Sobreflujo de complemento a 2, lo que facilita operaciones con signo. Esta bandera es usada para los saltos de instrucciones con signo como BGT, BGE, BLE y BLT.

H → Bandera de medio acarreo: La CPU pone en “1” esta bandera cuando ocurre un acarreo entre los bits 3 y 4 del acumulador, durante la ejecución de instrucciones **ADD** y **ADC**. La bandera H junto a la instrucción **DAA** es útil en operaciones **BCD** (*Binary Coded Decimal*), que facilitan el manejo de números decimales representados en el mundo binario.

I → Máscara de Interrupción global: Cuando se hace “1” deshabilita cualquier interrupción a la CPU (excepto la **SWI**), una vez en “0” mediante la instrucción **CLI**, permite que sucedan las interrupciones a la CPU.



La bandera I del CCR permite controlar las interrupciones en esquema no anidado.



Una vez el procesador sale del estado de reset, la bandera de Interrupción I inicia en 1, con lo que las interrupciones no son permitidas, hasta no ejecutar la instrucción **CLI** o algún procedimiento que borre esta bandera.

Al ocurrir una interrupción la bandera es automáticamente puesta en “1”, evitando el esquema de interrupciones anidadas.

- N** → Bandera de Valor Negativo: En “1” indica que el resultado de una operación aritmética o lógica produce un resultado negativo, “0” indica que el resultado es positivo.
- Z** → Bandera de Cero: En “1” indica que el resultado de una operación aritmética o lógica produce como resultado un cero.
- C** → Bandera de acarreo: En “1” indica que una operación de adición produce un acarreo del bit 7, o que una operación de resta produce un préstamo. También algunas operaciones lógicas, de rotación y de manipulación pueden mover el estado del bit.



Receta

En todos los procesadores la bandera de inhibición de interrupciones arranca inhabilitada. Recordar que se deben habilitar de forma explícita las interrupciones para que el sistema pueda reconocerlas y procesarlas.

2.2.2 Modos de direccionamiento Freescale HC(S) 08

Dependiendo de la forma como los datos son procesados, las instrucciones tienen diferentes esquemas de operación, los modos de direccionamiento son los siguientes:

Inherente: la instrucción trae de forma implícita el dato a operar o no lo requiere. Son instrucciones muy óptimas ya que solo ocupan 1 byte de memoria de programa y por consiguiente también son las más rápidas. Ejemplos de este modo de direccionamiento son las instrucciones:



El modo de direccionamiento hace referencia a la manera en que las instrucciones del programa son conducidas a través de los diferentes módulos del microcontrolador y entre éste y los periféricos.

CLRA	; borrar el AccA.
CLRH	; borrar el H.
TPA	; transferir registro CCR al AccA.
CLI	; borrar la máscara de interrupción.
DECA	; Decrementar AccA.
INCA	; incrementar AccA.

Inmediato: indica que el operando, requerido para la ejecución o argumento, se encuentra a continuación de la instrucción actual. Se indica este modo de direccionamiento por el signo #, que va luego de la instrucción. El valor inmediato podrá ser uno o dos bytes, dependiendo de la medida del registro involucrado en la instrucción.

Ejemplos de este modo de direccionamiento son:

LDA #\$30	; carga el AccA con el valor 0x30.
LDHX #\$F432	; carga el H con 0xF4 y el X con 0x32.
AIX #100T	;incrementa 100 al H:X.
AIS #-15	; Decrementa el SP en 16 bytes.

Directo: la instrucción tiene a continuación la dirección del dato el cual está en la zona directa del mapa de memoria, es decir entre \$0000 y \$00FF. Son instrucciones óptimas de acceso de datos debido a que asumen que la parte alta de la dirección del dato es \$00, lo cual elimina un byte de acceso. Ejemplos de instrucciones directas son:

LDA \$40	; carga el AccA con el contenido de la dirección 0x00040.
LDHX \$20	;carga el H con el contenido de la dirección 0x0020 y el X ; Con el contenido de la dirección 0x0021.
CMP \$30	;compara el contenido del AccA con el contenido de la ; Dirección 0x0030.
STA \$50	;almacena el contenido del AccA en la dirección 0x0050.

Extendido: las instrucciones extendidas son la generalización del modo Directo, en la cual la dirección del dato a operar está entre 0x0100 y 0xFFFF, todas las instrucciones de este modo son de 3 bytes de longitud, el primero es el opcode de la instrucción, el segundo la dirección alta de memoria y por último la dirección baja de memoria donde está ubicado el dato. Ejemplos de este modo de direccionamiento son:

LDX \$6E00	; carga el X con el valor que contiene la dirección 0x6e00.
LDA \$4000	; carga el AccA con el contenido de la dirección 0x4000.
AND \$8000	;realiza AND bit a bit del AccA con el contenido de la ;dirección \$8000
0x8000, resultado queda en AccA	



Relativo (+-128): este modo de direccionamiento analiza el resultado del opcode, el cual indica un salto corto que depende de su condición, un byte posterior (normalmente el segundo) indica en formato con signo el valor en que el PC es alterado si la evaluación es de la condición *es verdadera*. En caso que la condición sea falsa, el PC continuara con la siguiente instrucción. Las condiciones de salto examinan las banderas del CCR y con base en su estado (“1” o “0”) se tomará la decisión de salto o no.

En los procesadores Freescale HC, cuando se produce una interrupción, el modelo de programación es almacenado en el stack.



Ejemplos de uso de esta instrucción:

BEQ \$2F	;salta si es igual a Cero → PC+0x2F si ; La bandera de Z en "1".
BHI \$FE	; salta si IRQ en "1" → PC-1 si pin de IRQ en "1".
BMI \$0F	; salta si menor → PC+16 si bit de n en "1" en CCR.
BRCLR 1,\$40,\$10	;salta si el bit 1 del registro \$40 está en ;"1" → PC+\$10.
BRA \$0A	; salta siempre →PC+10.
BCC \$FD	; salta si C en 1 →PC-2 si bit de Acarreo C en "1".

Indexado: los modos de direccionamiento indexados usan el registro H: X para completar la dirección efectiva sobre la cual se operara, dependiendo del Offset que se use se pueden tener varios modos específicos de direccionamiento indexado:

Sin Offset: las instrucciones sin Offset son de 1 byte de longitud que accesan datos con direcciones variables. H: X contiene la dirección del dato a operar

Ejemplos

LDA ,x ;carga AccA con lo contenido en la dirección H:X.
STA ,x ;almacena el AccA en la dirección apuntada por H:X.
JMP ,x ;salta a la dirección apuntada por el registro H:X.

Offset de 8 bits: permite además adicionar un byte el cual se usa como Offset con respecto al H: X, es útil en tablas de datos, arreglos y matrices.

Ejemplos de uso:

JMP \$FF,x ; salta a la dirección apuntada por H: X mas \$FF.

Offset de 16 bits: es la generalización del Offset de 8 bits, en la cual el Offset puede ser de 2 bytes, con lo que se permite apuntar a cualquier dirección del mapa de memoria de 64K.

Ejemplo de su uso:

JMP \$10FF,x ; salta a la dirección apuntada por H: X + \$10FF.

Sin Offset, 8 bits con Post incremento: este direccionamiento es similar al indexado sin Offset y al indexado de 8 bits, pero además ofrece la posibilidad de agregar un "+" al registro X, que resulta en su incremento después de ejecutar la instrucción. Las instrucciones soportadas por este tipo de direccionamiento son: CBEQ y MOV.

Ejemplos de su uso:

CBEQ X+, Loop ; compara y salta si no es igual a cero.

Stack Pointer con Offset de 8 bits y 16 bits: este tipo de direccionamiento opera de forma similar a los direccionamientos indexados, con la diferencia que el registro usado es el SP y no el H: X.

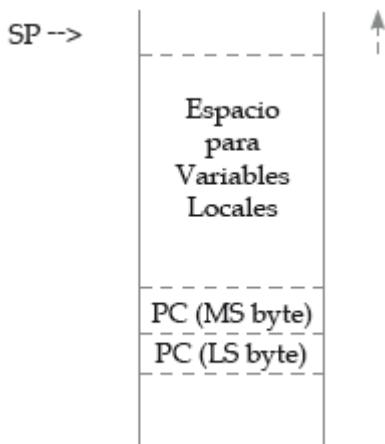
Ejemplos:

```
STA $10,SP      ; El AccA lo almacena en SP+0x10.  
LDA $2050,SP    ;carga el AccA con el contenido de la dirección  
                 ; SP+0x2050.
```

Este modo de direccionamiento permite el fácil acceso a datos que están guardados en el *stack* de forma temporal, en programación de alto nivel como el C, el manejo de variables declaradas como locales, así:

Función1

```
AIS #-16      ; crea 16 bytes de espacio local en RAM.  
...           ; Código de programa.  
AIS #16      ; recupera SP al valor inicial.  
RTS          ; retorna de subrutina.
```



Nótese con atención que el SP está apuntando siempre a la dirección siguiente disponible en el *stack*.

Memoria a memoria: este direccionamiento permite almacenar un dato de una fuente “*source*” a un destino “*dest*”, sin necesidad de ocupar los acumuladores para este fin.

El formato general es mov source, dest, y tiene las siguientes variantes:

Movimiento de valor inmediato a directo

Ejemplo:

MOV #20, \$40 ; Almacena en la dirección 0x40 un 20 decimal.

Movimiento de dirección directa a dirección directa

Ejemplo:

MOV \$20, \$40 ;almacena el contenido de la dirección 0x20 en la ; Dirección 0x40.

Movimiento de índice a dirección directa con post incremento

Ejemplo:

MOV X+, SCDR ;almacena lo apuntado por H:X en el registro SCDR, ;al terminar incrementa registro H:X.

Movimiento de dirección directa ha indexado con post incremento

Ejemplo:

MOV SCDR, X+ ;almacena el contenido del registro SCDR en la ;apuntada por H:X, al final incrementa el registro H:X.

2.2.3 Mapa de memoria del microcontrolador AP16A

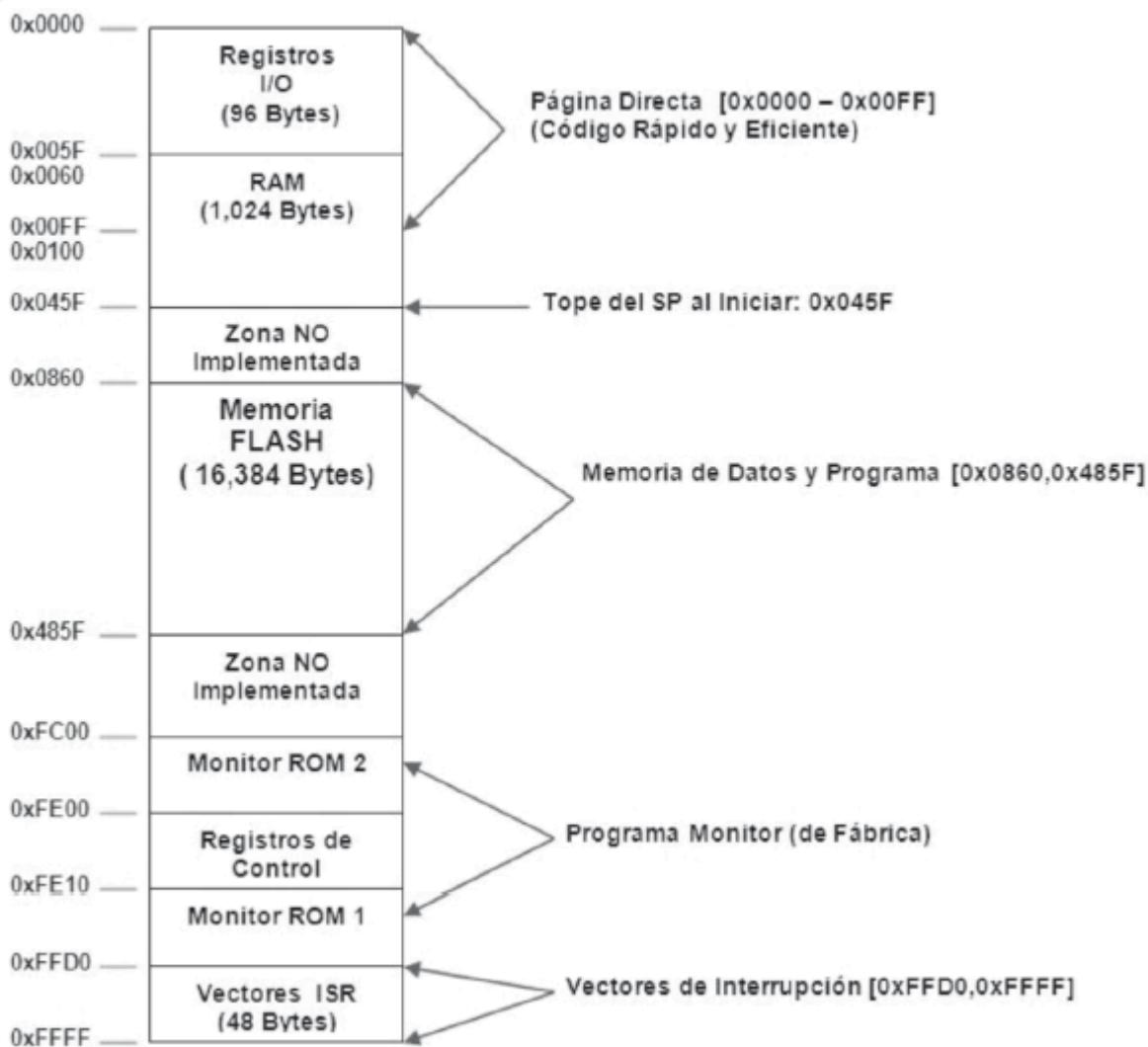
La CPU 08 tiene capacidad de direccionar un espacio de memoria de 64K, desde 0x0000 hasta 0xFFFF.

Esta ventana proporciona la visibilidad a los registros internos, a la memoria RAM, la memoria de programa, el programa monitor pregrabado de fábrica y los vectores de interrupción.

Existen algunas zonas que no están implementadas, por lo que cualquier acceso a ellos no proporcionara datos validos (*ver Gráfico 2.14*).

**GRÁFICO
2.14**

Mapa de memoria del microcontrolador MC68HC908AP16A de la familia HC08 de Freescale™.



2.2.4 Características del microcontrolador AP16A

El microcontrolador es uno de los miembros de bajo costo y alto desempeño de la familia de microcontroladores de 8 bits HC08 de Freescale™.

Las características principales de este microcontrolador:

Frecuencia de operación de 8 MHz interno en operación a 5V.

Opciones de oscilador RC o cristal oscilador de 1 a 8 MHz.

Memoria flash de programa de 16K bytes y memoria RAM de 1024 bytes.

2 Timers (TIM1 y TIM2) de 16 bits, con 2 canales cada uno configurable como entrada de captura, salida por comparación o PWM.

Módulo de base de tiempo (TBM).

Módulo de comunicaciones seriales SCI de propósito general (SCI1).

Módulo de comunicaciones seriales SCI con decodificador de IR (Infrarrojo) (SCI2).

Módulo de comunicaciones serial sincrónico (SPI).

Módulo de comunicaciones I2C.

ADC de 10 canales con 10 bits de resolución.

Interrupciones externas IRQ1 e IRQ2.

32 pines de configurables entrada/salida.

Módulos de protección COP (Watchdog), LVI, Opcode Ilegal y dirección Ilegal.

GRÁFICO
2.15

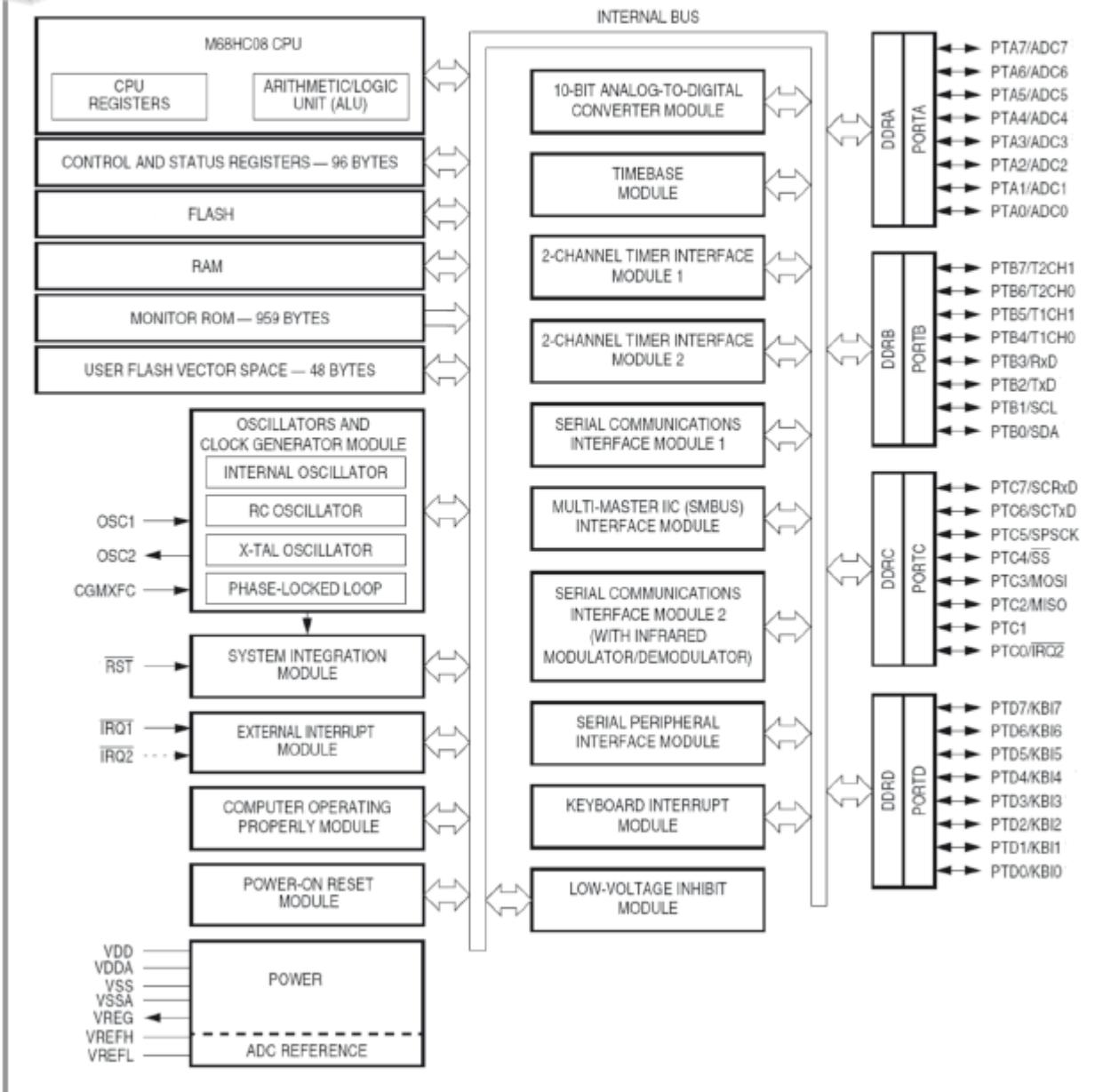
Distribución de pines del microcontrolador AP16A usado para el desarrollo de algunos de los ejemplos.

PTD2/KBI2	1	42	VDDA
PTD1/KBI1	2	41	VSSA
PTD0/KBI0	3	40	PTD3/KBI3
PTB7/T2CH1	4	39	PTD4/KBI4
CGMXFC	5	38	PTD5/KBI5
PTB6/T2CH0	6	37	PTD6/KBI6
VREG	7	36	PTD7/KBI7
PTB5/T1CH1	8	35	VREFH
VDD	9	34	VREFL
OSC1	10	33	PTA0/ADC0
OSC2	11	32	PTA1/ADC1
VSS	12	31	PTA2/ADC2
PTB4/T1CH0	13	30	PTA3/ADC3
TRQ1	14	29	PTA4/ADC4
PTB3/RxD	15	28	PTA5/ADC5
RST	16	27	PTA6/ADC6
PTB2/TxD	17	26	PTA7/ADC7
PTB1/SCL	18	25	PTC2/MISO
PTB0/SDA	19	24	PTC3/MOSI
PTC7/SCRxD	20	23	PTC4/SS
PTC6/SCTxD	21	22	PTC5/SPSCK

Diagrama de pines del microcontrolador MC908AP16ACB

Este diagrama es útil examinarlo cuando se está seleccionando el microcontrolador más adecuado para una aplicación particular, dependiendo del hardware externo que se requiere (*ver Gráfico 2.15*).

A continuación se presenta el diagrama de bloques del microcontrolador específico MC908AP16ACB de Freescale™ en el cual aparece por bloques los módulos de CPU, memoria FLASH para almacenamiento del programa, memoria RAM usada para los datos y *stack*, oscilador interno y módulo generador de frecuencia. Los módulos periféricos de este procesador son: El módulo **SPI**, el módulo de **SCI** y de **IrSCI**, módulo **I2C**, módulo **ADC**, módulo **LVI**, módulo temporizador y **TBM** (*Time Base Module*), y puertos de entrada/salida (I/O).

GRÁFICO
2.16Diagrama de bloques microcontrolador MC908AP16ACB¹.

2.2.5 Fuentes de interrupción del microcontrolador AP16A

Una fuente de interrupción que solicita atención de la CPU no suspende la instrucción que está siendo ejecutada, cuando la instrucción es completada el módulo interno del procesador verifica si existe alguna interrupción pendiente que esté habilitada, para proceder a atenderla.

Si más de una interrupción está pendiente al final de una instrucción, se ejecutará la que tiene mayor prioridad de acuerdo a la tabla de vectores de interrupción propia de cada procesador.

Base de tiempo

Para gestionar la prioridad de las interrupciones, el procesador usa la tabla de vectores de interrupción propia de cada procesador.

El módulo TBM genera interrupción de forma periódica dependiendo de la configuración realizada al momento de su configuración.

Esta fuente de interrupción es comúnmente usada como Reloj de Tiempo Real o RTC (*Real Time Clock*), por generar una interrupción periódica que no depende de los tiempos de ejecución y además por ser de bajo consumo.

Su funcionalidad como interrupción es controlada mediante el TBIE en el registro TBCR (Time Base Control Register).

Trasmisor serial SCI2 vacío

Genera interrupción cuando el dato registro de transmisión serial del SCI2 está vacío, indicando que el dato enviado por esta interfaz ya fue realizado y está listo para recibir un nuevo dato de una trama de datos.

El control de habilitación de esta interrupción particular se hace mediante el bit **SCTIE** en el registro SCC2 (*SCI Control Register 2*).

Receptor serial SCI2 lleno

Genera interrupción una vez que un dato nuevo está disponible en el registro de datos del interfaz serial del SCI2.

El control de habilitación de esta interrupción se realiza mediante el bit **SCRIE** en el registro **IRSCC2** (*SCI Control Register 2*).

Usado comúnmente en aplicaciones que tienen conexión con dispositivos externos para indicar que ya un dato llegó completo, sea por ejemplo una terminal de teclado externo, un sistema de localización o GPS (*Global Position System*) etc.

Error en interfaz serial SCI2

Genera interrupción por causa de algún error detectado en el interfaz IrSCI (SCI2), ya sea por error de overrun (*Overrun Error*), error de ruido (*Noise Error*), error de formato (*Framing Error*), error de paridad (*Parity Error*). La ocurrencia de esta interrupción de uno a más errores se realiza con los bits

ORIE (*Overrun Interrupt Enable bit*), NEIE (*Noise Error Interrupt Enable bit*), FEIE (Framing Error Interrupt Enable bit) y el PEIE (*Parity Error Interrupt Enable bit*), todos ellos en el registro IRS CC3 (*IRSCI Control Register 3*).

Trasmisor serial de SPI vacío

Genera interrupción una vez que el registro de transmisión serial sincrónico SPI se encuentra vacío, disponible a recibir un siguiente dato de una trama.

El control de interrupción se realiza mediante el bit SPTIE del registro SPCR (*SPI Control Register*).

Receptor serial de SPI lleno

Genera interrupción cada vez que se tiene un nuevo dato en el registro de datos del interfaz SPI.

El control de interrupción se realiza mediante el bit SPRIE en el registro SPCR (*SPI Control Register*).

Fin de conversión de ADC

Este módulo genera interrupción a la CPU cuando una conversión del ADC es realizada y el dato puede ser leído en el registro ADR (*ADC Data Register*).

El control de esta interrupción se realiza mediante el bit AIEN en el registro ADSR (*ADC Status and Control Register*).

Esta interrupción comúnmente se usa cuando al microcontrolador se conectan señales analógicas del mundo exterior como serían en un proceso industrial la presión de un tanque, la temperatura interior, nivel, humedad etc.

Módulo de interrupción tipo teclado

Genera interrupción por cambio en alguno de los pines del Puerto D (PORTD) del microcontrolador.

El control de forma independiente se realiza en los bits del registro KBIER.

Trasmisor serial SCI2 vacío (SCITx)

Genera interrupción cuando el dato registrado de transmisión serial del SCI1 está vacío, indicando que el dato enviado por esta interfaz ya fue realizado y está listo para recibir un nuevo dato de una trama de datos.

El control de habilitación de esta interrupción particular se hace mediante el bit SCTIE en el registro SCC1 (*SCI Control Register 1*).

Receptor serial SCI1 lleno (SCIRx)

Genera interrupción una vez que un dato nuevo está disponible en el registro de datos del interfaz serial del SCI1.

El control de habilitación de esta interrupción se realiza mediante el bit **SCRIE** en el registro **SCC2** (*SCI Control Register 2*)

Error en interfaz serial SCI1 (SCIError)

Genera interrupción por causa de algún error detectado en la interfaz IrSCI (SCI1), ya sea por error de overrun (*Overrun Error*), error de ruido (*Noise Error*), error de formato (*Framing Error*), error de paridad (*Parity Error*). La ocurrencia de esta interrupción de uno a mas errores se realiza con los bits ORIE (*Overrun Interrupt Enable bit*), NEIE (*Noise Error Interrupt Enable bit*), FEIE (*Framing Error Interrupt Enable bit*) y el PEIE (*Parity Error Interrupt Enable bit*), todos ellos en el registro SCC3 (*IRSCI Control Register 3*).

Interfaz del bus serial I2C

Genera interrupción por transmisión, recepción, pérdida de arbitración o ausencia de recepción de confirmación del modulo I2C.

El control de esta interrupción se realiza mediante el bit **MMIEN** del registro **MMCR1** (MMIIC Control Register 1).

Sobreflujo del Timer 2

Genera interrupción cada vez que el temporizador Timer 2 pasa de su valor máximo a cero (*overflow*).

El control de esta interrupción se realiza mediante el bit **TOIE** en el registro **T2SC** (*TIM2 Status and Control*).

Esta fuente de interrupción es comúnmente usada en aplicaciones que requieren una base de tiempo estable para realizar una labor de forma periódica, como puede ser tomar una muestra de una variable, tiempo de espera de una respuesta o una confirmación de una clave por parte del usuario etc.

Canal 1 del temporizador Timer 2

Canal 0 del temporizador Timer 2

Canal 1 del temporizador Timer 1

Canal 0 del temporizador Timer 1

Cada una de estas fuentes de interrupción genera interrupción cuando se presenta alguno de los eventos configurados para cada uno de los canales, ya sea: entrada por captura, salida por comparación o PWM.

Sobreflujo del temporizador Timer 1

Genera interrupción cada vez que el temporizador Timer 1 pasa de su valor máximo a cero (*overflow*).

El control de esta interrupción se realiza mediante el bit **TOIE** en el registro **T1SC** (*TIM1 Status and Control*).

Oscilador PLL

Genera interrupción cuando la frecuencia generada por el circuito oscilador pierde su valor programado. **PLLIE** en el registro **PCTL** (*PLL Control Register*)

Pin externo IRQ 2

Solicita atención de la CPU cada vez que el pin externo **IRQ2** cambia de estado, dependiendo de la configuración del flanco o nivel programado.

El control de esta interrupción lo realiza el estado del bit **IMASK2** del registro **INTSCR2** (*IRQ2 Status and Control Register*).

Pin externo IRQ 1

Solicita atención de la CPU cada vez que el pin externo **IRQ1** cambia de estado, dependiendo de la configuración del flanco o nivel programado.

El control de esta interrupción lo realiza el estado del bit **IMASK1** del registro **INTSCR1** (*IRQ1 Status and Control Register*).

Interrupción de software

Se genera una interrupción si la instrucción inherente **SWI** es encontrada en el código de programa que está ejecutando el procesador.

Esta interrupción es no enmascarable y es la única interrupción que se ejecuta independiente del estado de la bandera I en el CCR.

EJEMPLO No. 1 Encendido de Led en lenguaje de máquina

Objetivo:

El objetivo del primer programa consiste en desarrollar un código en lenguaje ensamblador que encienda el LED OUT-1 de prueba en el sistema de desarrollo AP-Link, cada vez que la tecla INPUT-1 es presionada.

Solución:

```
,***** Ejemplo 1 *****
; Encendido de Led en Lenguaje de Máquina
; Fecha: Feb 4,2009
; Asunto: Encender Led OUT-1 cada que la tecla
; INPUT-1 es presionada.
; Hardware: Sistema de desarrollo AP-Link (2008-01-14)
; para Microcontrolador Freescale AP16.
```

```

; Version: 1.0 Por: Gustavo A. Galeano A.
;*****
; Include derivative-specific definitions

INCLUDE 'derivative.inc'

; export symbols
XDEF _Startup
ABSENTRY _Startup
; variable/data section
ORG RAMStart ; Insert your data definition here
ExampleVar: DS.B 1
; code section
ORG ROMStart

_Startup:
LDHX #RAMEnd+1 ; initialize the stack pointer
TXS
CLI ; enable interrupts
mainLoop:
STA COPCTL ;reset al Watchdog
brclr 1,PTD,is_press ;PTD1 está en bajo?
bclr 3,PTC ; apaga OUT-1
bra mainLoop ; y salta al mainLoop
is_press: bset 3,PTC ; esta presionada, enciende OUT-1
bset 3,DDRC ;pin PTD3 como salida
BRA mainLoop ;salta al mainLoop
;*****
;* Interrupt Vectors *
;*****
ORG $FFFE
DC.W _Startup ; Reset

```

Discusión:

Para la creación de un programa en lenguaje de máquina, se inicializa el vector de Reset, ubicado en las direcciones \$FFFE y FFFF, con la dirección de inicio. En este caso una vez que el microcontrolador sale del estado de Reset, va a la función , _Startup que inicializa el Stack Pointer al final de RAM, esto lo hace con la ayuda del registro H:X y luego transfiriendo su contenido al SP con la instrucción TXS.

El código pasa luego a la sección mainLoop, donde está constantemente inicializando el contador del Watchdog que evita Reset, verifica si el pin PTD1 (pin 1 del puerto PTD) está en cero (tecla presionada) o en 1 (tecla no presionada), toma la decisión de saltar a la sección is_press, sí y solo sí el pin está en bajo donde pone el estado del PTC3 en 1 (enciende OUT-1) y además pone el pin PTC3 como salida con un estado 1 en el pin DDRC3.

Continúa el loop en mainLoop, verificado el estado del pin PTD1 y toma decisión según su estado, de tal forma que si el estado cambia, actualiza el estado de OUT-1.

RESUMEN DEL CAPÍTULO

Es de vital importancia, al programar en alto nivel un sistema embebido, conocer la arquitectura interna del procesador que correrá el código, debido a que en ocasiones la depuración puede llevar a requerir ejecutarlo paso por paso en lenguaje ensamblador.

Pudiera también ser necesaria la implementación de algunas rutinas o procedimientos en lenguaje de máquina, por querer realizar código más eficiente, o diseñar labores que el microcontrolador tenga que ejecutar y que desde el lenguaje de alto nivel sean más complicadas o no se tengan procedimientos para ellas.

La arquitectura Microchip™ está basada sobre la clasificación RISC de procesadores de alto desempeño por contener 2 buses internos, uno para los datos y otro para las instrucciones.

La arquitectura específica de Freescale™ ofrece un modelo lineal en mapa de memoria, donde se encuentran las direcciones de los registros internos que manejan los diferentes periféricos de cada microcontrolador en particular, la memoria RAM, la memoria de programa FLASH, y los vectores de Interrupción, todos direccionados con el PC y especificados al compilador de forma explícita.

PREGUNTAS Y EJEMPLOS SUGERIDOS

Mencione dos ventajas de la zona de acceso directo de un mapa de memoria, sobre la zona extendida.

¿Qué ventajas y desventajas puede tener en procesamiento de datos, usar registros de 16 bits, sobre los registros de 8 bits?

¿Cuál es la razón fundamental para posicionar estratégicamente los registros del microcontrolador en la página directa?

Realizar un código en lenguaje ensamblador que maneje 2 grupos de datos en forma BCD, sumando 1 y sumando 2 ocupan 4 bytes cada uno con valores entre 0 y 9. La rutina deberá sumar BCD y arrojar el resultado en 4 bytes consecutivos denominado resultado.

Realizar una rutina en lenguaje de máquina para HC08 que organice una tabla de datos de mayor a menor. La rutina recibe en el registro H:X la dirección de inicio de la tabla, y el AccA recibe la longitud de la tabla a organizar.

Enumere una ventaja de la arquitectura RISC sobre la CISC, y una de la CISC sobre la RISC.

INTRODUCCIÓN

No existe un compilador general sobre el cual se pueda trabajar para diferentes marcas de procesadores, cada una, y en algunos casos también cada familia de procesadores, tiene su propio compilador, de tal forma que es necesario tomarse el tiempo para navegar por el software y conocerlo, así será más rápido y fácil el desarrollo del código.

Este capítulo define qué es y qué elementos contiene un ambiente integrado de desarrollo (IDE) de compilador, establece algunos aspectos que deben considerarse una vez que se va a realizar la adquisición de un compilador, y particularmente se muestra el manejo de dos ambientes asociados a las arquitecturas de Microchip™ y de Freescale™: el CCS y el Codewarrior®; se describe paso a paso la creación de un proyecto en cada uno de los anteriores ambientes, y se realiza en cada uno un proyecto sencillo que va desde la concepción del problema hasta la edición del código, la compilación, la programación del chip y la depuración.

Se muestran también varios botones de acceso rápido y opciones de configuración de cada ambiente que agilizan la programación y establecen un esquema de trabajo sistemático y eficiente en el ciclo de diseño: editar, compilar, programar y ensayar.

También se enseñan algunas técnicas usadas en cada compilador para el manejo de las optimizaciones, tema importante cuando se programa una máquina con recursos de memoria de programa y RAM muy limitados,

El capítulo termina con un ejemplo práctico y sencillo sobre encendido de una salida que maneja un led usando lenguaje C; este proyecto, aunque básico, contiene todos los pasos que deben realizarse para la creación de los proyectos en capítulos posteriores; además, se ilustra el uso de cada botón involucrado desde la edición hasta la ejecución del código en el microcontrolador.

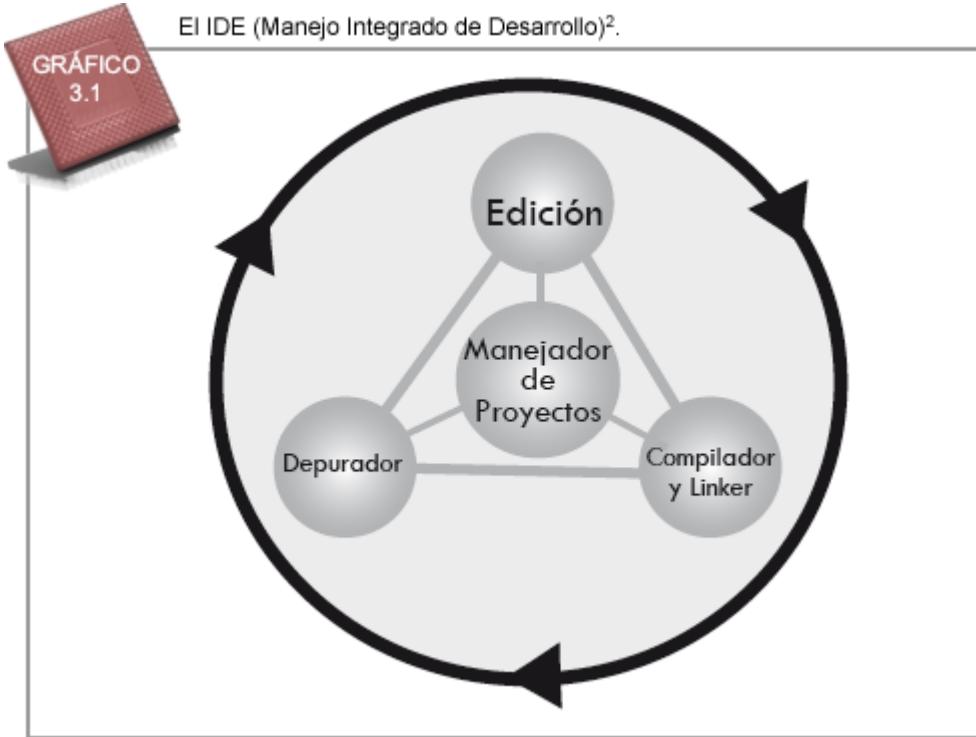
3.1 EL MANEJADOR DE PROYECTOS (IDE)



Dependiendo de la máquina seleccionada que ejecutará el código final, se seleccionará el respectivo ambiente de trabajo.

En general, se puede decir que un buen software tiene un **manejador de proyectos** llamado el IDE¹, el cual se encarga de administrar la **edición** del código, la **compilación** y la **depuración** (*Gráfico 3.1*).

El programador debe navegar y conocer el software compilador asociado a su respectivo procesador, ya que no existe un compilador universal, aunque todos guardan características comunes.



La trayectoria de la compañía diseñadora del software y el soporte técnico son dos aspectos a tener en cuenta al momento de elegir un compilador para lenguaje C.

En ocasiones puede ser de mayor utilidad un compilador de fácil manejo, que uno más robusto pero que exija un estudio

Existen en el mercado gran cantidad de herramientas de compilación, dependiendo del fabricante y la tecnología, se disponen desde herramientas de libre distribución, hasta algunos de costo moderado que permiten el objetivo básico de un compilador, que es pasar código en ANSI C a lenguaje de máquina ejecutable por el microcontrolador.

En este punto es muy importante conocer muy bien los siguientes aspectos antes de seleccionar una plataforma de desarrollo para un proyecto, debido a que para una máquina en particular existen muchos fabricantes.

Trayectoria de la compañía diseñadora: indagar un poco sobre la compañía que diseñó el software de desarrollo, lo cual está ligado con su permanencia en el mercado.

No sobra conocer si el compilador tiene certificaciones de buen desempeño: *Benchmarks*³.

Soporte técnico ofrecido: los compiladores pueden tener “bugs”⁴ en su funcionamiento, en su instalación etc. Acá es importante que

más detallado. El tiempo invertido en los compiladores complejos se justifica si el número de proyectos a desarrollar sobre esa base es alto.



se pueda acudir ya sea vía telefónica o por correo electrónico, al representante de la compañía encargado de asistir al usuario de forma regular.

Instalación en varios sistemas operativos: verificar cuáles son los sistemas operativos sobre los que se puede instalar el IDE: Windows, XP, Linux, Vista etc. Esto asegura que no se tendrá que cambiar o comprar otro compilador una vez cambie de máquina o de sistema operativo.

Fácil manejo: examinar la integración que tiene el editor con su respectivos compilador y depurador. Es deseable que se manejen de forma integrada bajo un mismo paquete. Examinar las bondades del editor, de las máquinas de búsqueda, de las opciones del compilador y la amigabilidad del depurador.

Costos: recordar: es mejor invertir un costo extra por un buen compilador. En este punto se debe examinar como es el esquema de licencias vs. costo del compilador, números de maquinas por licencia, vigencia de la licencia, costo de actualizaciones, vigencia del soporte técnico, costo extra por cambio de tecnología.

Soporte de Hardware: verificar las plataformas de hardware soportadas para depuración.

-
- 1 IDE: Integrated Development Environment.
 - 2 Cortesía FreescaleTM semiconductor.
 - 3 Benchmarks: pruebas de rendimiento y robustez de un software.
 - 4 Big: mal funcionamiento de un software en condiciones muy específicas.

3.2 INTRODUCCIÓN AL COMPILADOR CSS PARA MICROCHIP

La compañía CCS www.ccsinfo.com ofrece compiladores para los microcontroladores de Microchip™ serie PIC10, PIC12, PIC14 PIC16, PIC18 PIC24 y para los dsPIC® DSCs, su exclusividad con Microchip™ hace que esté muy optimizado para su arquitectura, garantiza buena permanencia en el mercado y precios muy asequibles.

La compañía ofrece varios compiladores dependiendo de la subfamilia de procesadores a usar: 12, 14, 16 y 24 bits, y también la plataforma en la cual se ejecutará, ya sea Windows o Linux.

La versión estándar y que se usará para ilustrar su uso y los ejemplos tratados es el **PCWH** que tiene un precio en el mercado de USD 500 para Windows (en la página de CCS es posible bajar una versión demostrativa sin costo para evaluación).

Este compilador incluye librerías estándar del ANSI C listas para invocar, varias rutinas para el manejo de los periféricos internos del microcontrolador además de varios drivers como RTC (Reloj de Tiempo Real), LCDs, Conversor A/D, retardos, manejo de puertos seriales etc.

El compilador PCD facilita la migración de 8 bits a procesadores de 24 bits con poca intervención del programador.

El compilador posee un IDE (*Integrated Development Environment*) completo, que integra el editor, la programación del código y depuración de la aplicación, lo que permite al programador diseñar, desarrollar, implementar y probar sus aplicaciones directamente en el procesador final (*target*).



Al seleccionar una tecnología para el desarrollo de productos embebidos, tener en cuenta aspectos que involucren costo, facilidad de manejo, soporte y flexibilidad.

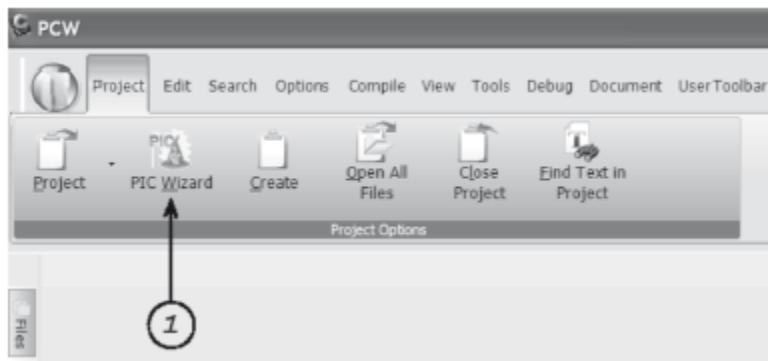


El compilador estándar de la compañía CCS para Microchip™ posee un IDE muy completo, el cual posibilita al programador la integración del editor, la programación del código y su depuración.

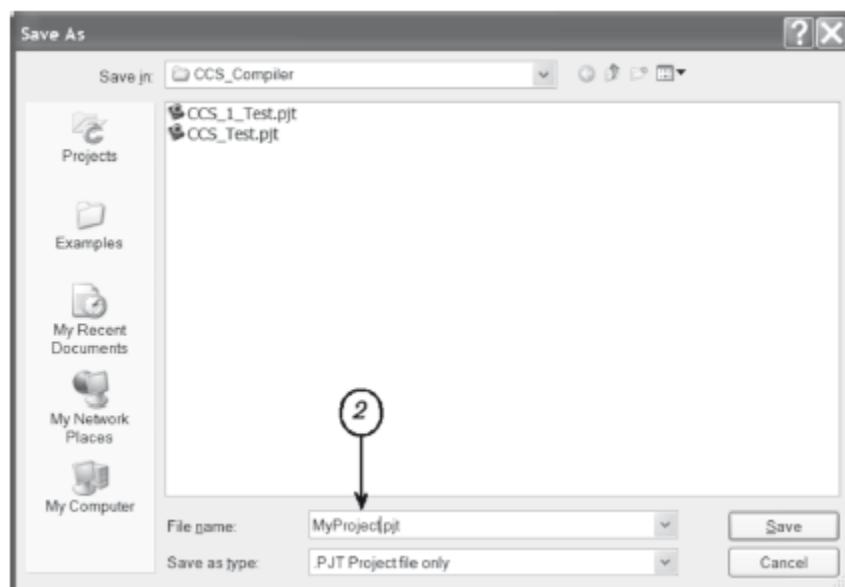


3.2.1 Creación de proyectos embebidos usando el ambiente CCS

1. **Inicio con PIC Wizard:** para iniciar un nuevo proyecto en el CCS se puede acudir a dos iconos, el de *Project → Create*, con el que se creará un proyecto de forma manual que contiene la información básica; la segunda opción, y recomendada por tener mayor contenido sobre el procesador específico y permitir parametrización, es por medio del ícono *Project → PIC Wizard*, paso 1.



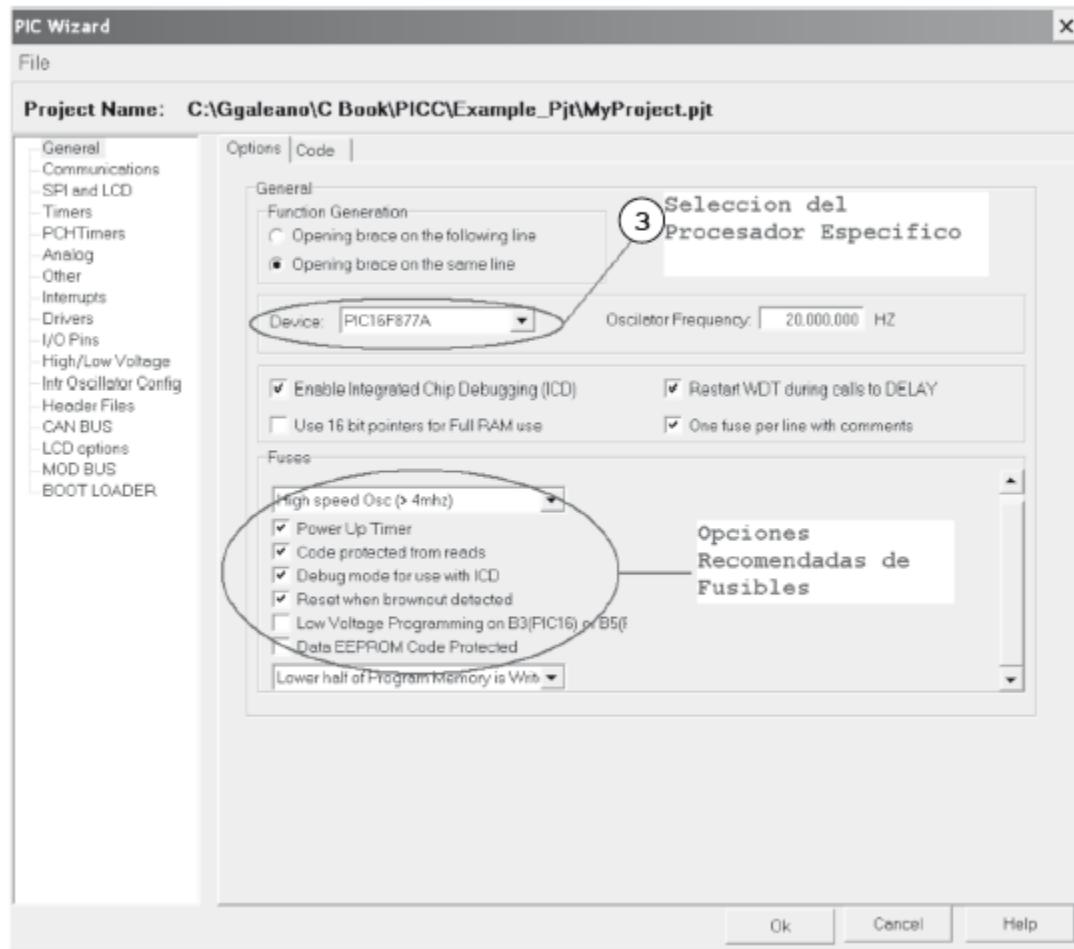
2. **Nombre del Proyecto:** en la siguiente ventana se debe digitar el nombre que tendrá el nuevo proyecto, y presionar el botón de “Save”, paso 2.



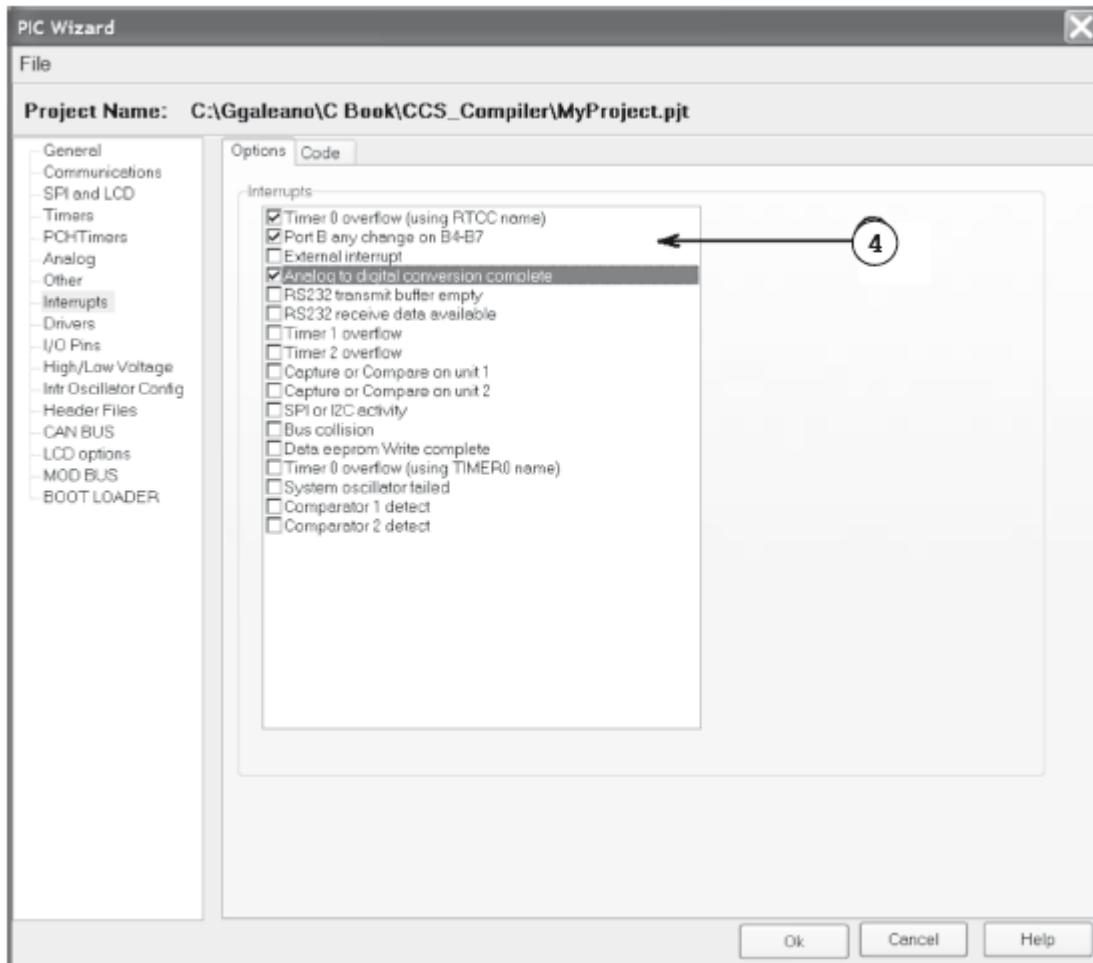
3. **Selección del Procesador:** la siguiente ventana en aparecer solicita el procesador específico seleccionado para el proyecto.

NOTA: los ejemplos de esta publicación estarán desarrollados usando el PIC16F877A, como se muestra en la siguiente ventana.

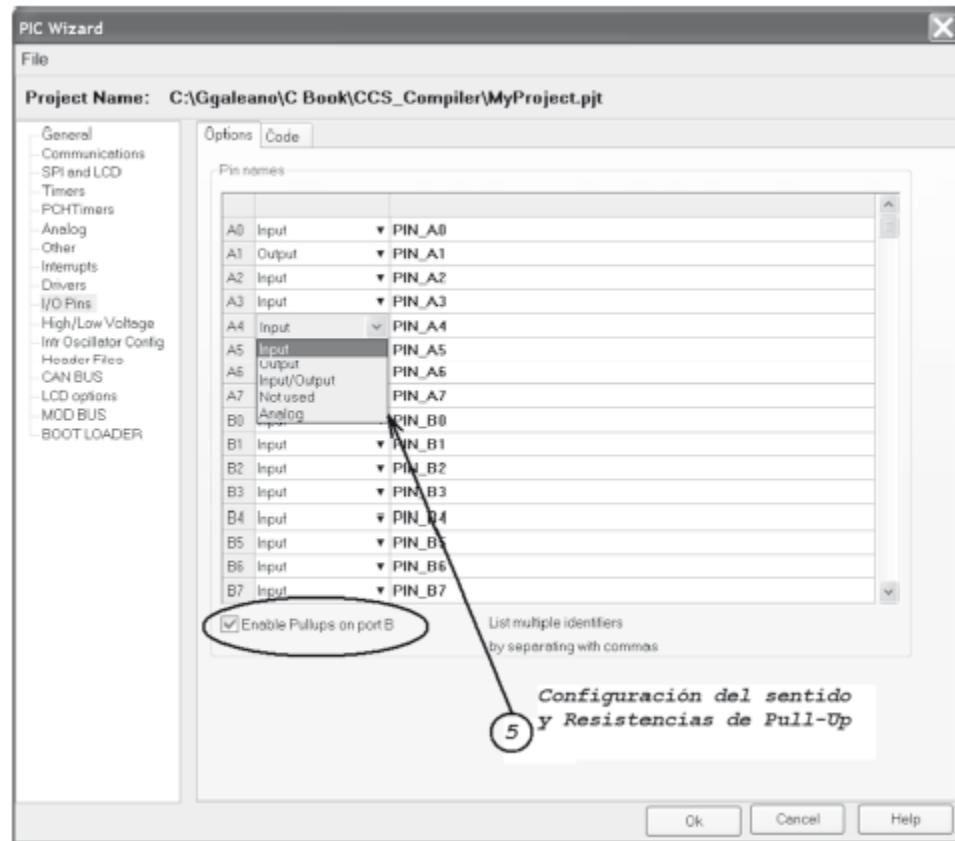
También permite seleccionar los botones que protegen el código, tanto de memoria de programa como los contenidos de EEPROM, para que no puedan ser leídos.



4. **Selección de Interrupciones:** permite seleccionar las interrupciones que serán usadas en la aplicación, con el fin de adicionar el código base y el enunciado de las funciones al archivo principal del proyecto.

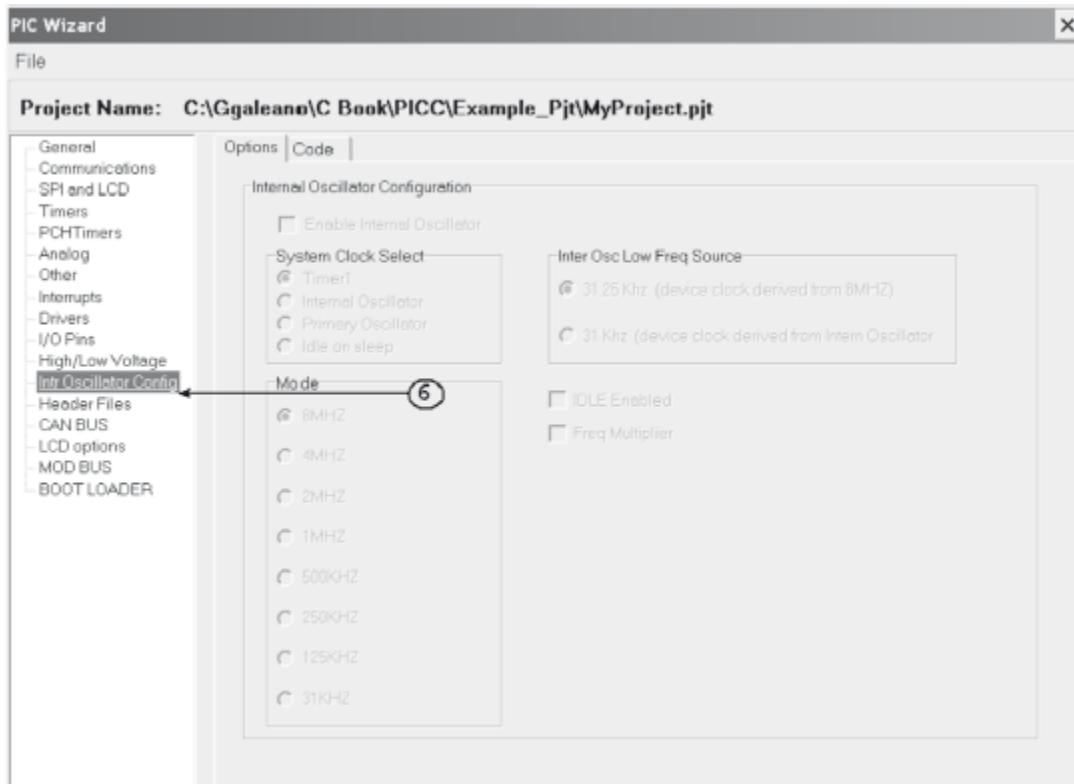


5. **Sentido de cada pin:** se elige el sentido que tiene cada uno de los pines del microcontrolador seleccionado, también se tiene posibilidad de habilitar o deshabilitar las resistencias de *pull-up*⁵ disponibles en el puerto B.



6. **Opciones de Oscilador Interno:** en caso de usar el oscilador interno del microcontrolador permite seleccionar la frecuencia de operación.

En el caso de los ejemplos publicados se usa un cristal externo que se seleccionó en el paso 3.



Al presionar el botón OK se tiene el proyecto creado y se mostrará la ventana de editor con el archivo principal del proyecto.

```

1 #include "C:\Ggaleano\C Book\CCS_Compiler\MyProject.h"
2 #int RTOC
3 void RTOC_isr(void) {
4 }
5
6 #int RB
7 void RB_isr(void) {
8 }
9
10 #int AD
11 void AD_isr(void) {
12 }
13
14 void main(){
15     port_b_pullups(TRUE);
16     setup_adc_ports(MO_ANALOGS|VSS_VDD);
17     setup_adc(ADC_OFF);
18     setup_spi(SPI_SS_DISABLED);
19     setup_timer_0(RTOC_INTERNAL|RTOC_DIV_1);
20     setup_timer_1(T1_DISABLED);
21     setup_timer_2(T2_DISABLED,0,1);
22     setup_comparator(MC_NC_NC_NC); // This device COMP currently not supported by the PICWizard
23     enable_interrupts(INT_RTOC);
24     enable_interrupts(INT_RB);
25     enable_interrupts(INT_AD);
26     enable_interrupts(GLOBAL);
27     setup_oscillator(OSC_4MHZ);
28     // TODO: USER CODE!!
29 }

```

- 5 Pull-Up: resistencias con conexión del pin a fuente positiva.

3.2.2 Familiarización con el IDE del CCS

Editor del CCS



El editor de código del CCS resalta las palabras reservadas con un color propio e incluye control de tabulación y búsqueda de correspondencias de paréntesis y corchetes.



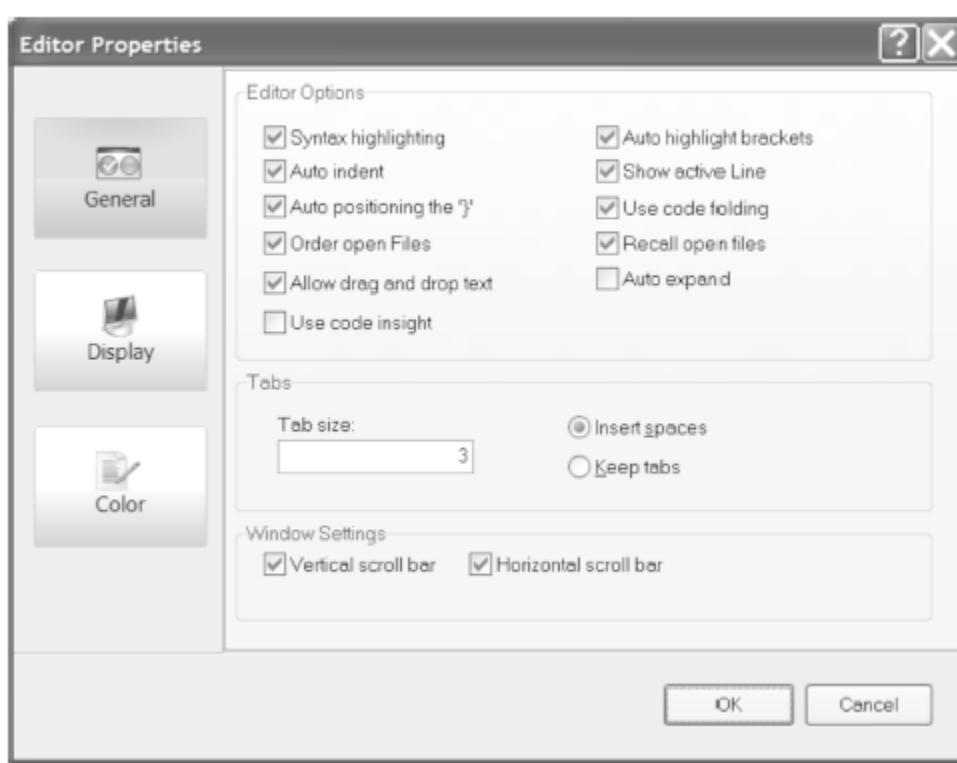
El editor del CCS contiene reconocimiento de la sintaxis del C, las palabras reservadas son resaltadas en un color propio, contiene control de tabulación, búsqueda de correspondencia de paréntesis, y corchetes. Permite además un fácil acceso al árbol de funciones, mapa de símbolos, así como a los programadores y depuradores.

El motor de búsqueda posibilita ubicar símbolos en los archivos abiertos o en los del proyecto completo.

Las opciones recomendadas para el desarrollo de proyectos usando el CCS se muestran en el *Gráfico 3.2*.

GRÁFICO
3.2

Opciones recomendadas en la ventana de propiedades.



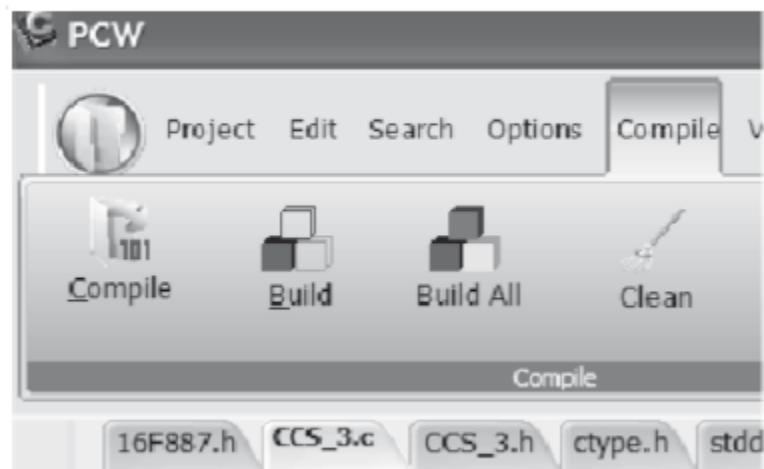
Compilador del CCS

La compilación del proyecto se realiza con el acceso a los botones *Compile* → *Compile* o bien *Compile* → *Built All* cuando se quiere compilar todos los archivos del proyecto (ver Gráfico 3.3).

La forma rápida y más directa de hacer la compilación es mediante la tecla **F9**.



Botones de compilación del CCS.



Una vez se realiza la compilación, en la parte inferior aparecerá la ventana de *Output*, que muestra los *Warnings* y *Errores*⁶ identificados en la compilación.

Luego de la compilación sin errores se obtienen como salida varios archivos con el nombre del proyecto y extensiones:

.ERR → WARNINGS y ERRORES, contiene el resultado de la ventana OUTPUT al compilar, resume los warnings y errores encontrados en el proyecto. También presenta un resumen del porcentaje de memoria ocupado por el proyecto, tanto en memoria de programa ROM (Flash en el caso del PIC16F877A) como memoria RAM.

>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 626(1,1): Condition
>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 633(1,1): Condition
>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 685(1,1): Condition
>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 694(1,1): Condition
>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 745(1,1): Condition
>>> Warning 203 "C:\Program Files\PICC\drivers\stdlib.h" Line 756(1,1): Condition
Memory usage: (ROM=4% (RAM=6%) - 6%
0 Errors, 6 Warnings.

Annotations: A callout bubble points to the 'Warning' column with the text 'Listado de Warnings'. Another callout bubble points to the memory usage percentages with the text 'Porcentaje de Memoria Ocupado por el proyecto'.



.LST → **LISTADO**, contiene equivalencia entre el código digitado en C y el correspondiente en lenguaje de máquina.

Útil cuando se requiere verificar la conversión de lenguaje C hacia el lenguaje de máquina, para evaluar la optimización de algún procedimiento antes de realizar la depuración.

Receta

Al realizar la compilación de un programa sin errores, se debe poner atención además a los **WARNINGS** (precauciones), que pueden indicar problemas potenciales en el software.

```
16F887.h CCS_3.c CCS_3.h ctype.h stddef.h stdio.h stdlib.h string.h
2479 ..... void Delay(void){
2480 .....     static unsigned int i;
2481 *
2482 00E9: CLRF 2E
2483 .....     for(i=0;i<200;i++) {
2484 *
2485 007B: CLRF 2E
2486 007C: MOVF 2E,W
2487 007D: SUBLW C7
2488 007E: BTFSS 03.0
2489 007F: GOTO 082
2490 ..... }
2491 0080: INCF 2E,F
2492 0081: GOTO 07C
2493 ..... }
2494 0082: RETLW 00
2495 }
```

.HEX → **HEXADECIMAL**, contiene un archivo de texto con el lenguaje de máquina generado, listo para ser programado en la microcontrolador definitivo.

El formato .HEX es estándar y puede ser usado por cualquier herramienta de programación que soporte tecnología Microchip™ y el procesador específico.

16F887.h CCS_3.c CCS_3.h CCS_3.lst ctype.h stddef.h stdio.h stdlib.h string.h TA CCS_3.hex

Address	00	01	02	03	04	05	06
0668	100C	118A	120A	2837	120D	118A	120A
0670	110B	118A	120A	2837	1683	1386	1283
0678	118A	120A	2938	01AE	082E	3CC7	1C03
0080	0AAE	287C	3400	1683	1386	1283	1786
0088	*****	*****	*****	*****	0083	3067	00F7
0090	*				0BF7	2892	01F8
0096	007B: CLRF 2E				01FE	3050	00F7
00A0	007C: MOVF 2E,W				0BF7	28A2	3060
00A8	007D: SUBLW C7				0AB4	0BF7	28AB
00B0	007E: BTFSS 03.0				1783	0180	0A84
00B8	007F: GOTO 082				1383	301F	0683
00C0	*****				0099	3002	009A
00C8	0080: INCF 2E,F				1683	1703	0809
00D0	0081: GOTO 07C				3000	1703	0088
00D8	*****				00FA	1303	0805
00E0	0082: RETLW 00				39FD	0087	1283



.SYM → MAPA DE SIMBOLOS, es un archivo de solo lectura que contiene la localización en memoria de cada registro y las variables de programa usadas en cada sección de programa.

16F887.h CCS_3.c CCS_3.h CCS_3.lst ctype.h stddef.h stdio.h stdlib.h string.h CCS_3.sym

10	021	@INTERRUPT_AREA
11	022	@INTERRUPT_AREA
12	023	@INTERRUPT_AREA
13	024	@INTERRUPT_AREA
14	025	@INTERRUPT_AREA
15	026	@INTERRUPT_AREA
16	027	@INTERRUPT_AREA
17	028-029	strtok.save
18	02A-02D	Randseed
19	02E	Delay.i
20	02F	@TRIS_C
21	077	@SCRATCH
22	078	@SCRATCH
23	078	RETURN
24	079	@SCRATCH
25	07A	@SCRATCH
26	07B	@SCRATCH
27	07F	@INTERRUPT_AREA

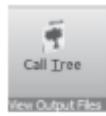


.COF → CODIGO DEBUG, contiene un archivo en formato texto con información estándar para depuración, no contiene información útil para el programador.

```

16F887.h CCS_3.c CCS_3.COF.btx CCS_3.h Ia CCS_3.hex CCS_3.sym ctype.h stddef.h stdio.h stdlib.h string.h
1 Coff-Dump Utility V1.2
2 Copyright (c) 2003 Custom Computer Services, Inc.
3
4
5 CoffFile (C:\Ggaleano\C Book\CCS_Compiler\CCS_3.cof):
6
7 File Header:
8 f_magic=4660
9 f_nscns=3
10 f_timdat=1224961597
11 f_symptr=0x000016F8
12 f_nsyms=531
13 f_opthdr=16
14 f_flags=0x0003

```



.TRE → Contiene el mapa de herencia de las funciones dentro del proyecto. Es útil para visualizar los procedimientos que cada módulo de programa invoca, permitiendo visualizar de forma global la estructura modular del proyecto.

Item	Page	ROM	
CCS_3			
MAIN	0	178	0
??0??			
Led_ON	0	7	0
Delay	0	8	0
Led_OFF	0	7	0
Delay	0	8	0
RTCC_isr	0	4	0
EXT_isr	0	4	0
AD_isr	0	4	0
TIMER1_isr	0	4	0
EEPROM_isr	0	4	0
TIMER0_isr	0	4	0



El archivo .STA
(Estadísticas)
del compilador CCS
ofrece información
sobre el uso de las

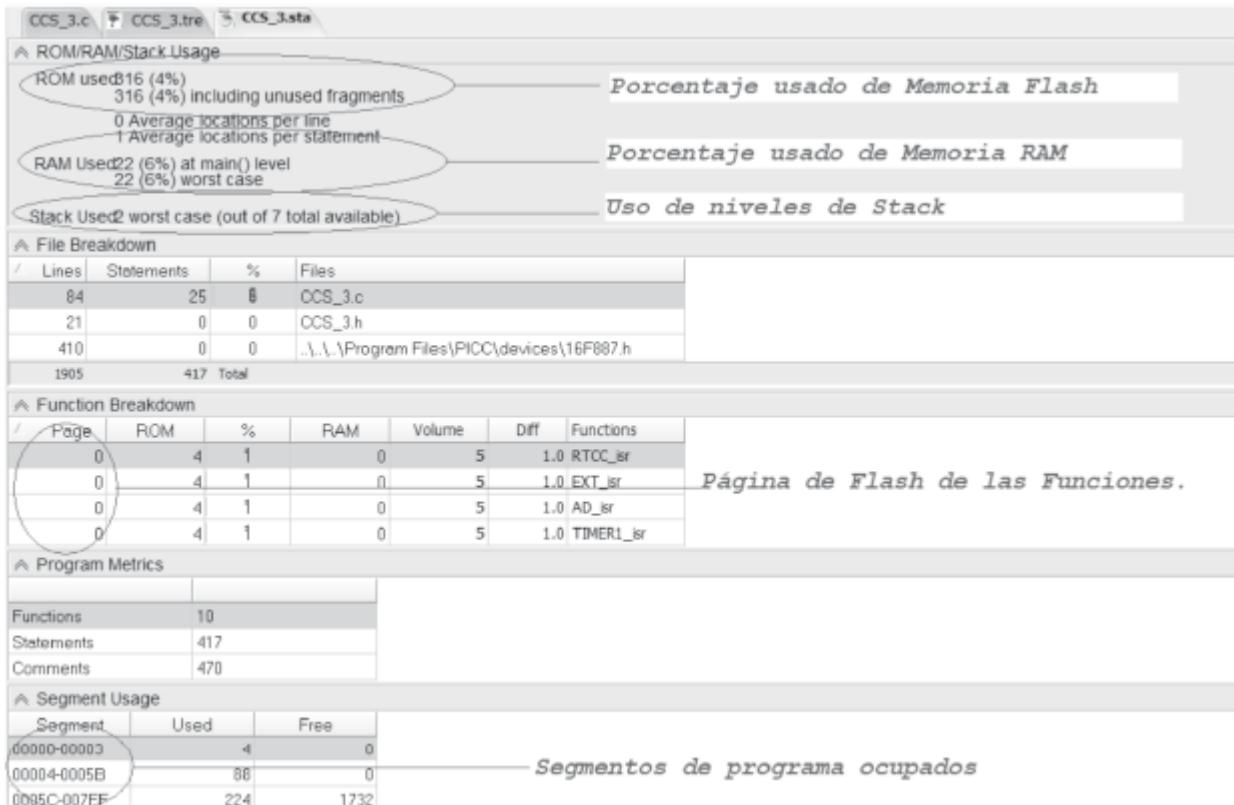


.STA → ESTADÍSTICAS, es un archivo gráfico que muestra de forma resumida, el uso en porcentaje de las memorias de programa flash y RAM, de forma similar a la presentada en el archivo de salida .ERR, indicando, además, el número de niveles usado del stack en el peor de los casos; este nivel corresponde al número de rutinas anidadas que tenga el programa + 1 (llamado a una interrupción), siempre y cuando no se permitan interrupciones anidadas.

memorias flash y RAM, discrimina los archivos y funciones y orienta al programador al proporcionarle una idea sobre la longitud del programa.

En un segundo y tercer bloque, se presenta una discriminación de todos los archivos y funciones indicando el número de líneas y uso específico en porcentaje de memoria usada.

La sección de métricas del programa (*Program Metrics*), presenta un resumen del número de funciones total del proyecto, además de 5 medidas de complejidad desarrolladas por *Halstead*, los cuales proporcionan una idea de la longitud del programa, su dificultad, esfuerzo, diversificación de operadores y mantenibilidad.



Depurador del CCS

El depurador del CCS provee la capacidad de enviar el archivo compilador ejecutable (.HEX) al microcontrolador y ejecutar el código en modo depuración (conectado al PC) o independiente (*standalone*), en el que el microcontrolador solo requiere la alimentación de voltaje.

Para habilitar los botones del depurador deberá conectarse el ICD-U40 al puerto USB y el sistema de evaluación PIC16F877A en el otro extremo, una vez alimentada la tarjeta, presionar el botón

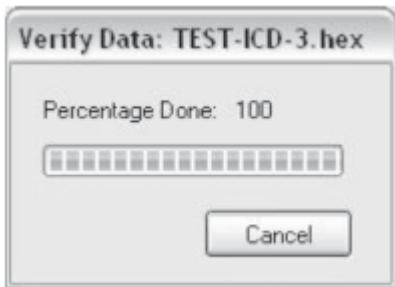


Los archivos de salida del CCS, proporcionan información útil al programador, que permiten evaluar la estructura del



, aparecerá entonces una ventana emergente con el mensaje “connecting to ICD”, y una ventana indicando el porcentaje programado, hasta llegar al 100%.

proyecto y el espacio ocupado en memoria de programa y RAM.

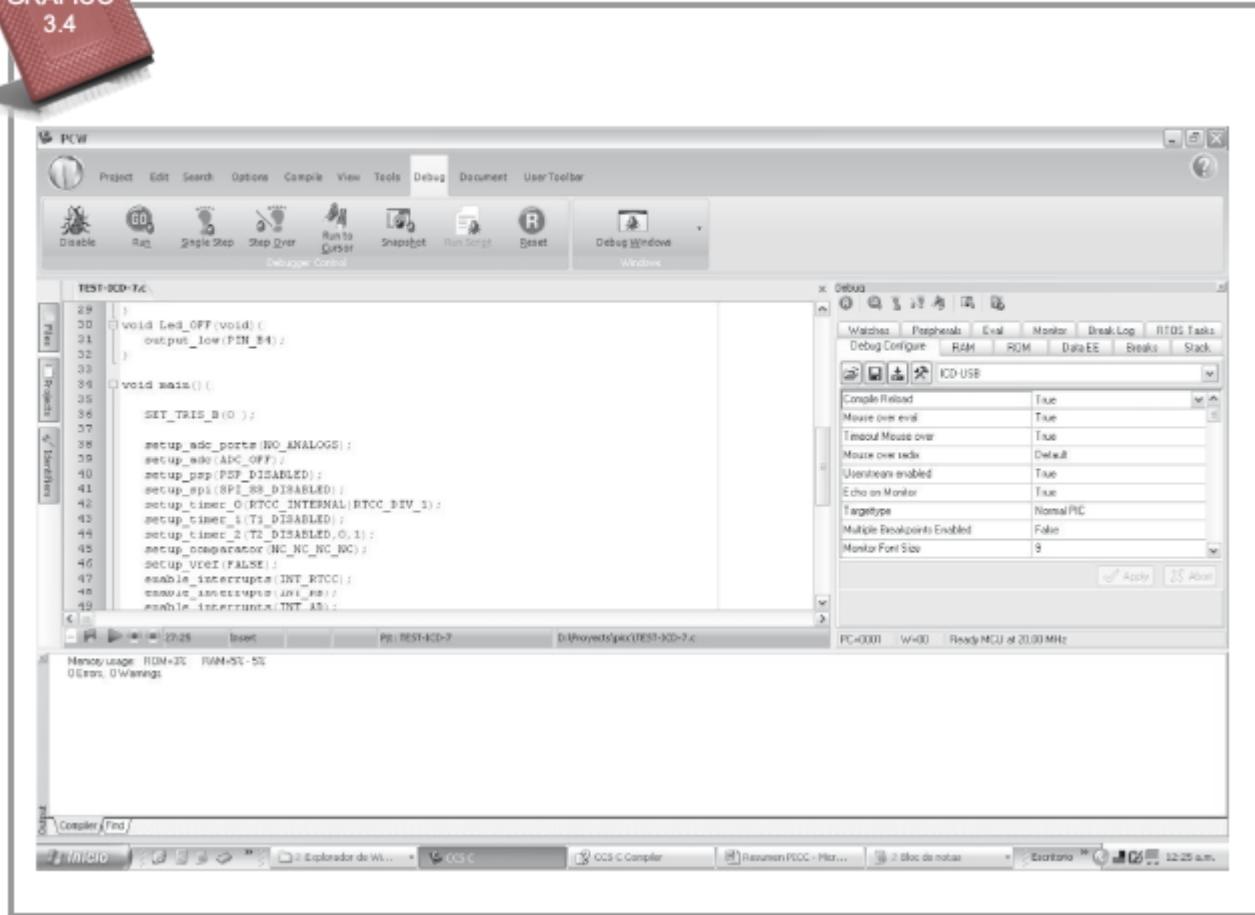


En este punto se puede ejecutar el programa en el microcontrolador, o bien realizar depuración del código; para realizarla, se accesa al icono: , una nueva ventana se abrirá:





Ventana de depuración del Compilador CCS.



Para la depuración se tiene acceso a los siguientes iconos que están disponibles en la parte superior de la ventana:



Single Step, ejecuta una línea de código de la ventana activa, permite entrar a depurar dentro de cada función.



Step Over, ejecuta una función de código completa del código fuente.



Run to Cursor, ejecuta el código del programa, desde el valor del PC actual, hasta la posición actual del cursor, una vez en este punto, actualiza el valor de las ventanas del depurador.



Snapshot, permite la grabación de información varia del depurador, el usuario elige qué información deberá grabarse.



Run Script, abre un programa de script que puede controlar el depurador. Permite realizar pruebas automatizadas y repetitivas usadas en procesos de producción.



R *Reset*, genera un *reset* en la tarjeta de evaluación, el PC es posicionado en su valor inicial e inicia ejecución desde este punto, hasta encontrar un punto de ruptura (*breakpoint*).



G *Run*, ejecuta el código del programa desde la posición actual del PC, las ventanas de depuración NO son actualizadas con la información actual, el programa ejecutará hasta encontrar un punto de ruptura o hasta presionar el botón de Halt.



S *Halt*, suspende la ejecución del programa en el sistema de evaluación, las ventanas de depuración son actualizadas con el valor actual de cada registro/ variable.

El control de entrada y salida de los pines digitales se realiza mediante las funciones incluidas en las librerías del compilador:

SET_TRIS_x(valor) → Define el sentido de los pines del PUERTO x.

OUTPUT_x(valor) → Define el valor digital de los pines del PUERTO x que estén definidos como salida.

INPUT_x() → Entrega como resultado el valor de entrada de los pines del PUERTO x del microcontrolador.

En el caso del PIC16F877A x puede tomar los valores A, B, C, D o E, que corresponden a los identificadores de cada puerto.

El manejo de los puertos por pines de forma independiente se realiza con las siguientes dos funciones, dependiendo si se requiere establecer el valor del pin en bajo (LOW) o en alto (HIGH):

OUTPUT_LOW(nombre_pin)

OUTPUT_HIGH(nombre_pin)

Los valores de *nombre_pin* están definidos en el archivo **16F877A.h** para este caso específico de microcontrolador.

El Primer programa en C usando compilador CCS

EJEMPLO No. 2

Encendido de led Microchip™ en C

Objetivo:

El objetivo del primer programa ANSI C es desarrollar un código que encienda y apague el LED OUT-1 en el PIN_B4 del puerto B de prueba en el sistema de evaluación PIC-Link.

Usar una función de retardo que permita verificar el funcionamiento en modo RUN.

Solución:

El primer paso será entonces crear un nuevo proyecto que se llamara **LED_TEST**, se crea el proyecto siguiendo uno a uno los pasos de creación de un proyecto nuevo:

1. Inicio con PIC Wizard.
2. Nombre del proyecto (LED_TEST.pjt).
3. Selección del procesador (Seleccionar PIC16F877A).
4. Selección de interrupciones (ninguna).
5. Sentido de cada pin (PIN B4 como salida).
6. Opciones de oscilador (ninguna).

Edición:

Una vez creado el proyecto se abre el archivo Led_Test.c en el editor, y se posiciona en la función void main(void).

Ahora y según el diagrama de conexiones del sistema de evaluación, el LED de prueba 1, llamado VISUAL OUT-1 encenderá con un nivel “1” en el pin B4 del Puerto B del Microcontrolador PIC16F877A, lo cual se hace con la línea:

```
output_high(PIN_B4); // poner un nivel “1” en B4 del Puerto B
```

Se realizan 2 macros, uno para encendido y otro para apagado del LED OUT-1 así:

```
#define Led_ON() output_high(PIN_B4)  
#define Led_OFF() output_low(PIN_B4)
```

El código del primer programa se verá de la siguiente forma:

Ejemplo 2:

```
// Encendido de Led Microchip en C  
// Fecha: Feb 4,2009  
// Asunto: Encendido y apagado de LED OUT-1
```

```

// usando Compilador CCS.

// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)

// para Microcontrolador Microchip™ 16F877A.

// Version: 1.0 Por: Gustavo A. Galeano A.

//*****



#include "C:\Ggaleano\C Book\PICC\Led_Test\Led_Test.h"

// Definición de Macros para manejo del pin del LED

#define Led_ON() output_high(PIN_B4)

#define Led_OFF() output_low(PIN_B4)

// Funcion de Retardo

void Delay(void){

static unsigned long i;

for(i=0;i<40000;i++){

}

}

// Funcion principal

void main(void){

port_b_pullups(TRUE);

setup_adc_ports(NO_ANALOGS);

setup_adc(ADC_OFF);

setup_psp(PSP_DISABLED);

setup_spi(SPI_SS_DISABLED);

setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);

```

```
setup_comparator(NC_NC_NC_NC);

setup_vref(FALSE);

// TODO: USER CODE!!

for(;;){

    Led_ON();

    Delay();

    Led_OFF();

    Delay();

}

}
```

Se finaliza la edición del programa.

Compilación:

La compilación se realiza presionando la tecla **F9**.

El compilador no deberá mostrar errores de sintaxis en la ventana de salida, de no ser así se debe verificar que el código está bien digitado comparándolo con el código mostrado.

Programación y Depuración:

Realizar ahora la conexión del ICD-U40 al puerto USB y al sistema de evaluación PIC-Link, como se muestra en el Gráfico 3.5.

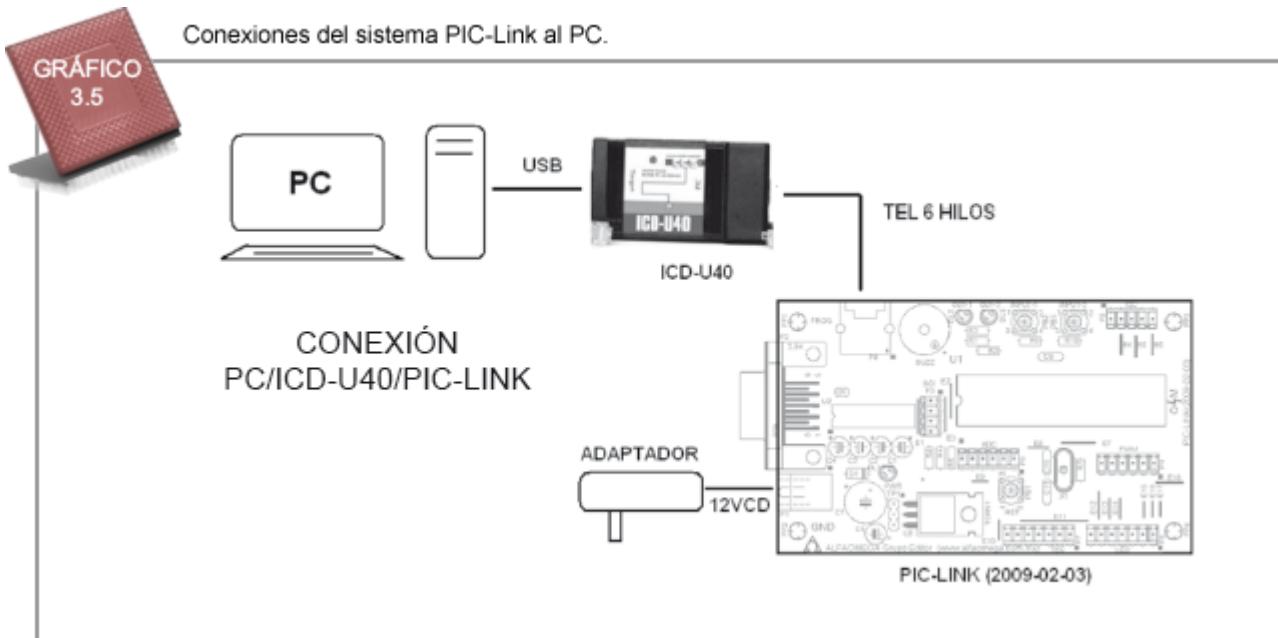
Una vez programado el código en el microcontrolador, este quedará residente en la memoria de programa.

Si se desea realizar depuración del programa que permite ejecutarlo pasó por paso, se puede acceder al ícono de depuración y usar los iconos correspondientes para verificar línea a línea la ejecución del programa.

Discusión:

El programa realizado fue editado y compilado, y quedará residente en la memoria flash del microcontrolador, se ejecutará automáticamente si el microcontrolador es energizado y sin dependencia del PC.

Para evaluar esta operación, se debe desconectar el ICD-U40 del sistema PIC-Link, al alimentar la tarjeta el LED deberá encender y apagar de forma visible, indicando que **el primer programa en ANSI C** está siendo ejecutado.



-
- 6 Warnings: Precauciones, posibles fuente de error en ejecución.



Materiales adicionales en la
Video explicativo sobre encendido de Led Microchip.

3.3 INTRODUCCIÓN AL COMPILADOR CODEWARRIOR® DE FREESCALE™



Para el caso particular de Freescale™, se cuenta con una herramienta bastante completa, llamada Freescale™ Codewarrior®, consiste en un ambiente completo IDE (*Integrated Development Environment*), o ambiente integrado de desarrollo conformado básicamente por los siguientes elementos: editor, manejador de proyectos, ensamblador, máquina de búsqueda, compilador Enlazador o *Linker*, depurador (en alto y bajo nivel).



Estos están ligados bajo un mismo ambiente que permite una operación consistente, de fácil movimiento entre uno y otro.

El compilador Codewarrior® (V6.2) cuenta con una versión libre, para procesadores de 8 bits y cuyo ejecutable no sobrepase los 32 K bytes.

El compilador de Codewarrior® y todo su ambiente V 6.2, está en versión libre para los procesadores de 8 bits para programas cuyo ejecutable (.S19) no sobrepase los 32K Bytes totales de capacidad para los procesadores de la familia HC(S)08, y 64K para los procesadores Coldfire V1, con lo cual se puede soportar gran capacidad de aplicaciones del sector universitario e industrial.

La versión puede bajarse con facilidad de la página www.freescale.com/codewarrior y solicitar la licencia especial de 32K vía e-mail license@freescale.com.

3.3.1 Creación de proyectos embebidos en C usando Codewarrior



siguiente ventana:

La creación de un proyecto prepara el ambiente del compilador, para, sobre este, incorporar los archivos fuentes (.C) y los de cabecera (.H); además, define el microcontrolador específico necesario para el enlazado y el tipo de depuración que se realizará, ya sea simulación en el mismo computador que realiza la compilación, o si el ejecutable será enviado al microcontrolador y allí ejecutado.

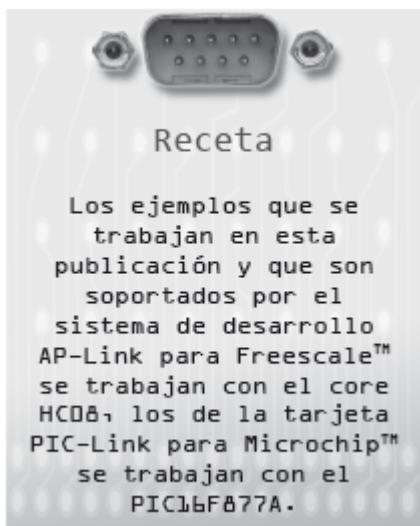
Para la creación de un proyecto en el ambiente que se ha seleccionado para esta publicación con Codewarrior® requiere de 10 pasos, los cuales serán mostrados a continuación:

1. **New Project:** al iniciar el Codewarrior® mostrara la



Elegir el botón “**Create New Project**”. Esta ventana puede configurarse para que no aparezca al inicio, deshabilitando la opción “**Display on Startup**”. Si esta ventana no es mostrada al inicio debido a esta configuración, este paso puede ser obviado e ir directamente al menú “**File**” → “**New Project...**” y pasar al punto 2.

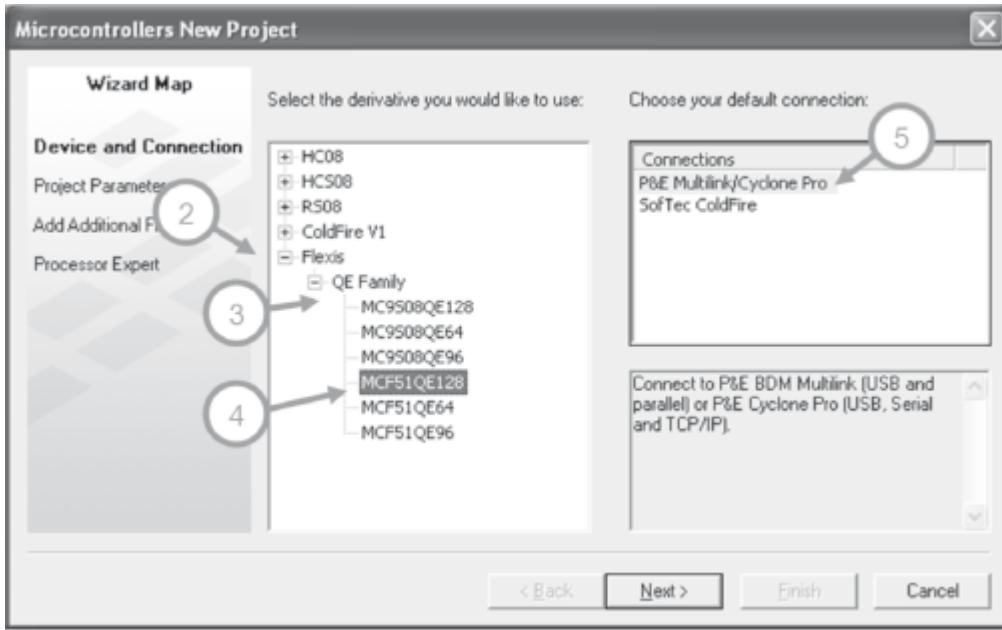
2. **Definición de Core:** elegir esta opción seleccionando el tipo de familia que va a usar para el proyecto. Las opciones disponibles en el Codewarrior® 6.2 son:



- a. **HC08** incluye los procesadores MC68HC908.
- b. **HCS08** que soporta los procesadores serie MC9S08 de tecnología de bajo voltaje de alimentación y bajo consumo de energía.
- c. **RS08** el cual cubre la gama más baja de Freescale™, MC9RS08KA, que son procesadores de pocos pines (6 y 8 pines) y de core CPU08 reducido en instrucciones.
- d. **ColdFire V1** cubriendo los procesadores de 32 bits V1 de gama más baja, estos son procesadores de alto desempeño y pin a pin compatibles con algunos de los microcontroladores MC9S08 de 8 bits.
- e. **Flexis.** Esta opción considera una alternativa adicional de seleccionar algún procesador de 8 ó 32 bits compatibles pin a pin, procesadores MC9S08QE y MCF51QE.

3. **Definición de subfamilia:** una vez seleccionado el *core*, se define la subfamilia del procesador a usar, en la mayoría de los casos esta corresponde al identificador que sigue el *core* en su número de parte: MC68HC908XX, MC9S08XX donde XX especifica la subfamilia.

Esta definición precisa el número de pines del procesador usado, los diferentes periféricos que contiene la parte internamente y el tipo de procesador dependiendo de la aplicación.



4. **Definición de la parte específica:** esta ventana solicitará el número de parte específico que determinará la parte a trabajar en el proyecto, define a su vez la capacidad de memoria de programa (Memoria Flash), y de manera implícita, la capacidad de RAM.
5. **Definir el tipo de Conexión para Simulación:** en esta ventana se define además el tipo de simulación que se usará una vez que el programa sea compilado y se genere el código ejecutable. De forma general existe la opción: “**Full Chip Simulation**”, en la cual toda la depuración se hace usando el mismo computador que se utilizó para la edición y compilación del proyecto, y también está la opción: “**In Circuit**” (o P&E), en la cual toda la depuración se realizará en el procesador definitivo de la aplicación.

En lo posible se debe tratar que la depuración sea “In Circuit”, para obviar cualquier problema asociado al hardware de la aplicación. La opción “Full Chip Simulation” es recomendada solo para aquellas secciones de programa o rutinas que solo sean de procesamiento de datos y que no tengan interacción con el hardware, de esta forma la depuración será más rápida debido a que no es necesario programar la flash del microcontrolador cada vez que se realizan cambios en el código, y requerirá de menos recursos y conexiones a la hora de evaluar código.

Presionar “Next” para pasar a la siguiente ventana.

6. **Nombre del proyecto:** en esta ventana se elige el nombre que llevará el proyecto y su ubicación. En esta selección es importante resaltar que el nombre debe ser tal que sea de fácil reconocimiento una vez se ha creado el proyecto, también es útil si se tiene un control de versiones de proyecto o backup del mismo proyecto. Un buen método es usar

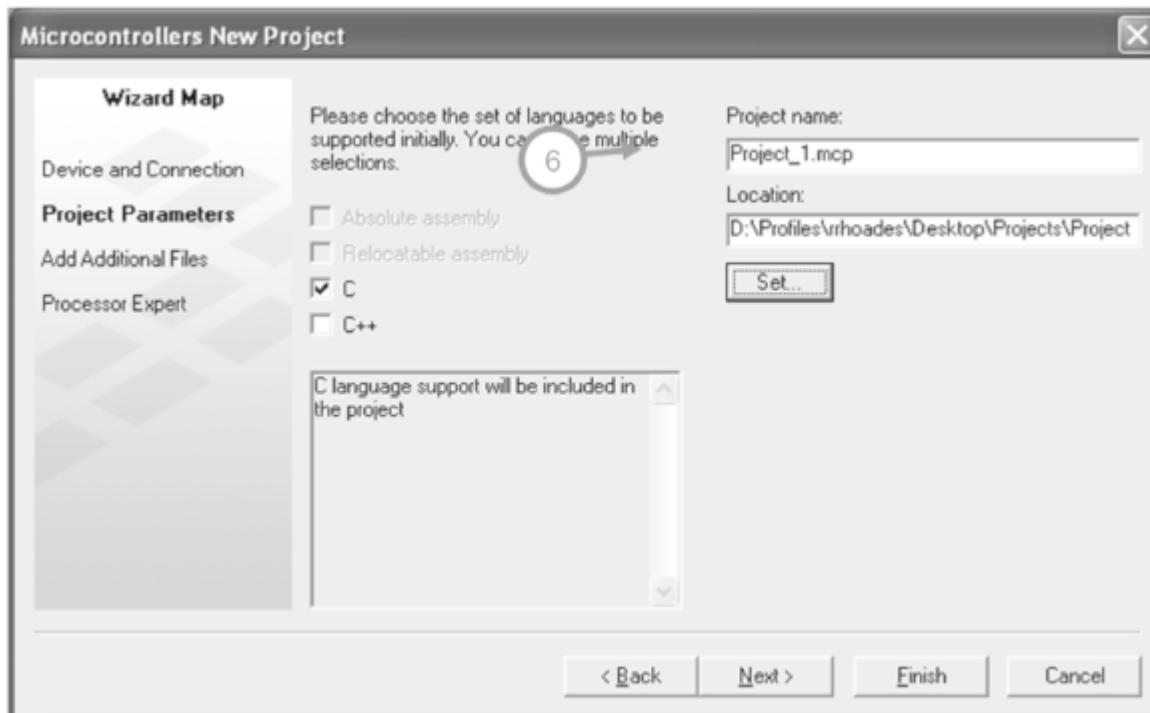


Los proyectos desarrollados en este texto usan la opción de lenguaje soportado C, que es la opción por defecto en la creación de cualquier proyecto; sin embargo es posible en el Codewarrior® trabajar también en lenguaje ensamblador gozando de todas las ventajas del ambiente integrado.

números consecutivos sobre el mismo nombre del proyecto sin cambiarle de nombre para ubicarlo con facilidad.



En algunos casos también resulta útil hacerlo utilizando alguna marca de fecha. Así por ejemplo se puede tener en el mismo directorio del proyecto varias versiones llamadas: Proyecto_1.mcp, Proyecto_2.mcp, Proyecto_3.mcp, Proyecto_4.mcp o de igual forma: Proyecto_241207.mcp, Proyecto_251207.mcp, Proyecto_271207.mcp.

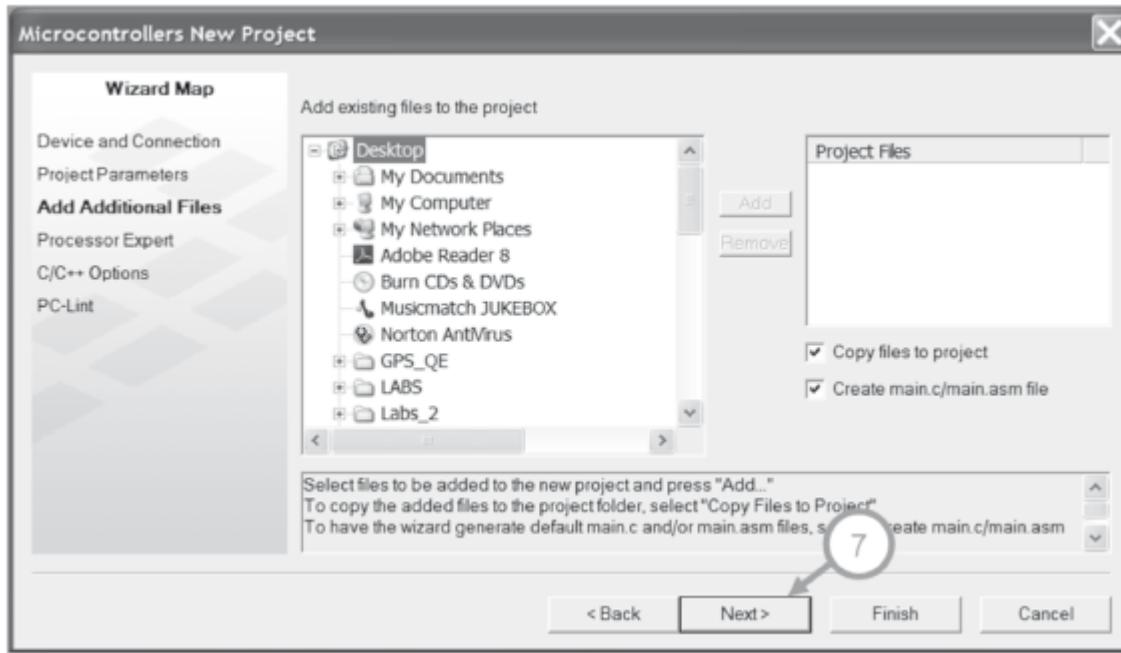


Para los proyectos desarrollados en este texto se usará la opción de lenguaje soportado C, que es la opción por defecto en la creación de cualquier proyecto, sin embargo es posible en el Codewarrior® trabajar también en lenguaje ensamblador gozando de todas las ventajas del ambiente integrado.

Las dos opciones de ensamblador se diferencian en que la opción de “**Absolute assembly**” genera código que está ligado a ejecutarse en determinadas direcciones de memoria, mientras que el “**Relocatable assembly**” está orientado a generación de código que podrían ejecutarse en cualquier dirección de Flash o RAM dado que el código es relocalizable, es decir, ejecuta el mismo código independiente de la dirección donde éste sea movido; esta opción es útil para generar código que estando en flash, en algún momento en tiempo de ejecución, se decide que debe ejecutarse en la RAM.

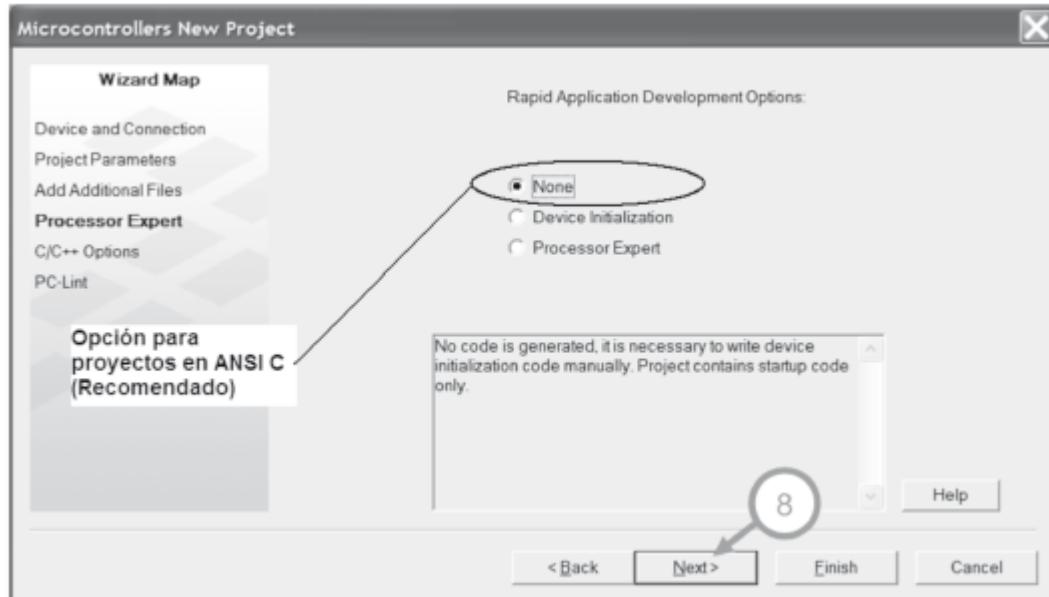
Presionar “Next” para ir a la siguiente ventana.

7. **Add Additional Files:** esta ventana permite adicionar al proyecto archivos existentes en C o en ensamblador al proyecto creado. Si no se requiere esta adición, presionar “Next” para ir directamente a la ventana siguiente.



8. **Uso de “Processor Expert™”:** esta ventana permite la configuración de opciones de inicialización y de uso del “Processor Expert™” de Codewarrior® 6.2, este tema está desglosado en el Capítulo 9.

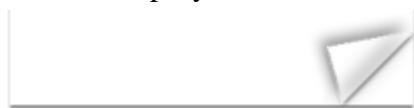
Para los ejemplos de ANSI C puro y tratados sin el “Processor Expert™” se usará la opción por defecto de esta ventana, que es la opción “None”. Con esta selección, el proyecto quedará creado solo con la función de inicio “*Startup*” y la invocación directa de la función “*main()*”, donde el programador iniciará su código.



Presionar la opción “Next” para continuar.



Es importante que el código generado en la compilación sea 100% compatible con el Ansi C, por eso no se recomienda excluir el código *Startup* del proyecto.



9. Opciones de C/C++:

en esta ventana se eligen tres opciones importantes para la generación de código:

1. La primera de ellas es la que invoca la generación del código de arranque “*Startup*” antes de la función “main()”. La opción marcada por defecto es la llamada “**ANSI startup code**” la cual deberá incluirse para garantizar que el código es 100% compatible con el Ansi C; en éste se establece que las variables globales son inicializadas en su valor al momento de la definición o en valor cero, en caso que no sean inicializadas.

Esta inicialización toma a veces muy poco tiempo, pero si el código de programa es extenso este tiempo es considerable y el arranque del procesador puede tardarse de forma considerable para el programa.

Solo en algunos casos muy especiales, en los cuales se requiere que el microcontrolador arranque de forma rápida su procesamiento, se recomienda usar la opción de “**minimal startup code**”, con ella solo es inicializado el SP (*stack pointer*) y se realiza el llamado a la función main() de inmediato. Es de anotar que las variables en RAM NO serán inicializadas y tendrán valores aleatorios, por lo que el programador deberá realizar esta inicialización a lo largo de su ejecución.

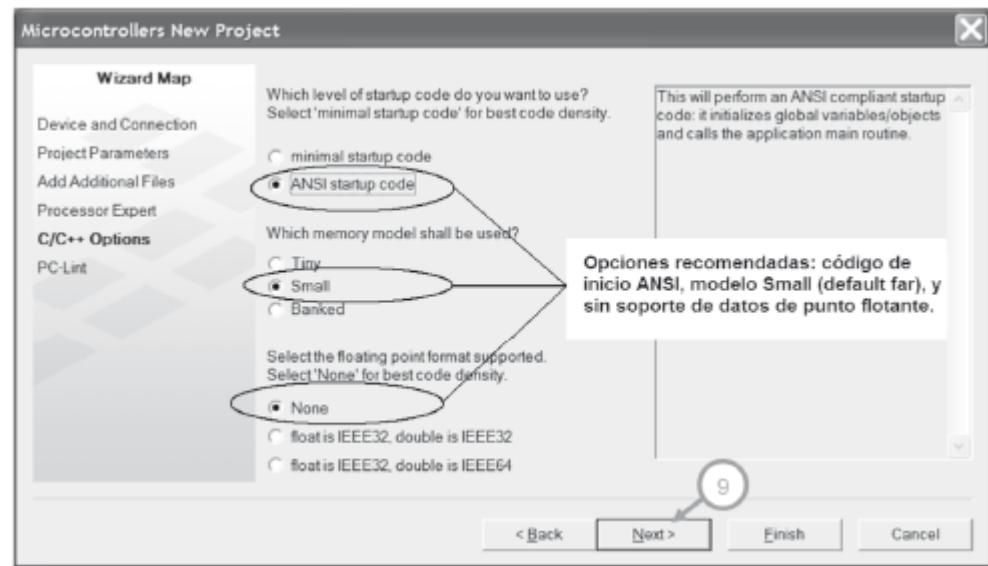
2. La segunda opción elige el modelo de memoria RAM a usar, en este caso se recomienda utilizar el modelo por defecto del Codewarrior® “Small” con el cual se define que todas las variables declaradas en RAM, estarán por fuera de la pagina cero (o directa) del mapa de memoria.

El lector podrá remitirse al Capítulo 5 (Modificadores de Variables), donde se define y se explica de forma precisa el significado y uso de las variables tipo “**near**” y “**far**”.

El modelo “*Banked*” es considerado para microcontroladores con bancos de memoria de programa, donde el mapa de memoria ocupa más de 64K.

Todos los ejemplos en esta publicación no sobrepasan o usan procesadores mayores a 64K, con lo que esta opción no deberá considerarse.

3. Permite la selección del tipo de punto flotante, en lo posible usar la opción por defecto “None”, en la cual no se cuenta con este soporte. Otras dos opciones son especificadas dependiendo del tipo de formato a usar, sea IEEE32 ó IEEE64.

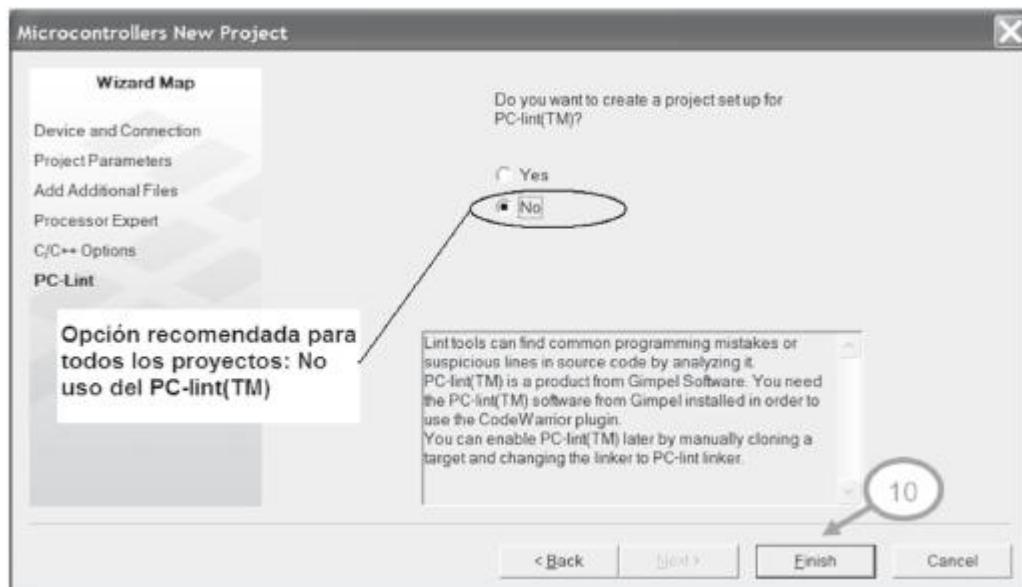


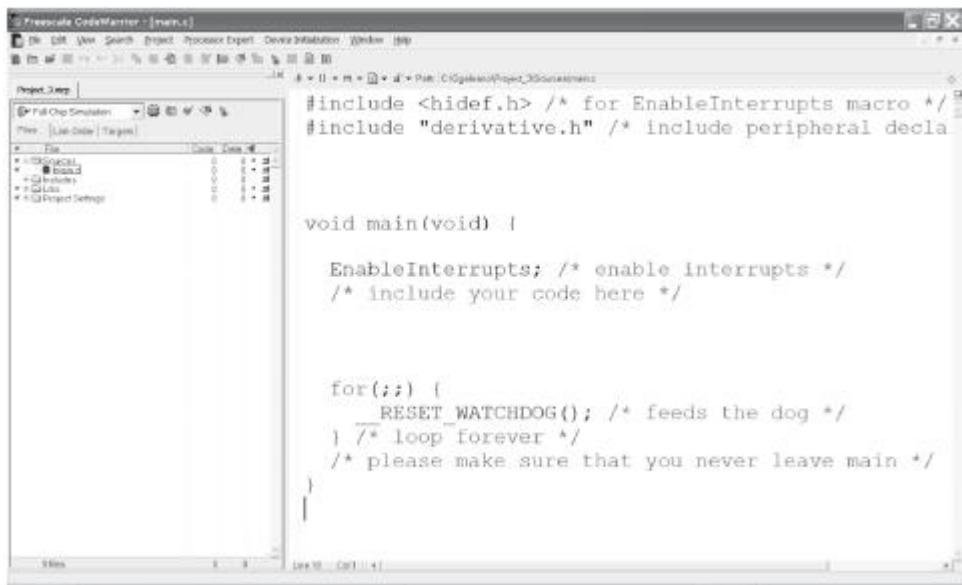
Presionar “Next” para continuar.

10. **Uso del PC_Lint:** esta opción permite elegir el uso del software externo llamado PC-Lint™ de la compañía Gimpel Software, mediante el cual se puede hacer depuración en tiempo de ejecución, sin necesidad de parar la ejecución del software del microcontrolador con el uso de una aplicación externa ejecutándose en el PC, y un código residente en memoria del microcontrolador.

Es recomendable usar la opción que viene por defecto “No” para optimizar el uso de memoria en el microcontrolador y además porque en esta publicación no se realiza depuración usando dicha herramienta.

Presionar el botón de “Finish” para terminar la creación del proyecto.





```
#include <chidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral decla

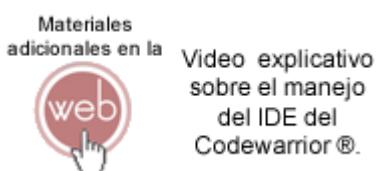
void main(void) {
    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    for(;;) {
        RESET_WATCHDOG(); /* feeds the dog */
    } /* loop forever */
    /* please make sure that you never leave main */
}
```

Una vez se tenga la ventana del proyecto, se puede acceder con doble *click* en el archivo main.c, y allí se puede empezar a editar el código.

3.3.2 Familiarización con el IDE de Codewarrior®

El Editor de Codewarrior®



Esta parte del ambiente constituye la herramienta para que el diseñador inserte su código. El editor de Codewarrior® es uno de los más amigables y completos, sin embargo si el usuario lo considera apropiado o tiene mejor manejo, pudiera hacer la edición en cualquier otro editor de texto, en dicho caso deberá recordar el guardar los archivos modificados antes de realizar la compilación del proyecto.

El editor de Codewarrior® está bien orientado al ANSI C, el cual maneja código de colores diferente para cada grupo editado, uno para las palabras reservadas del C, los comentarios, las cadenas (*strings*), las constantes, las funciones, las definiciones de tipo (*typedef*); adicionalmente, permite definir un grupo de palabras especiales del diseñador las cuales se resaltarán con un color definido.

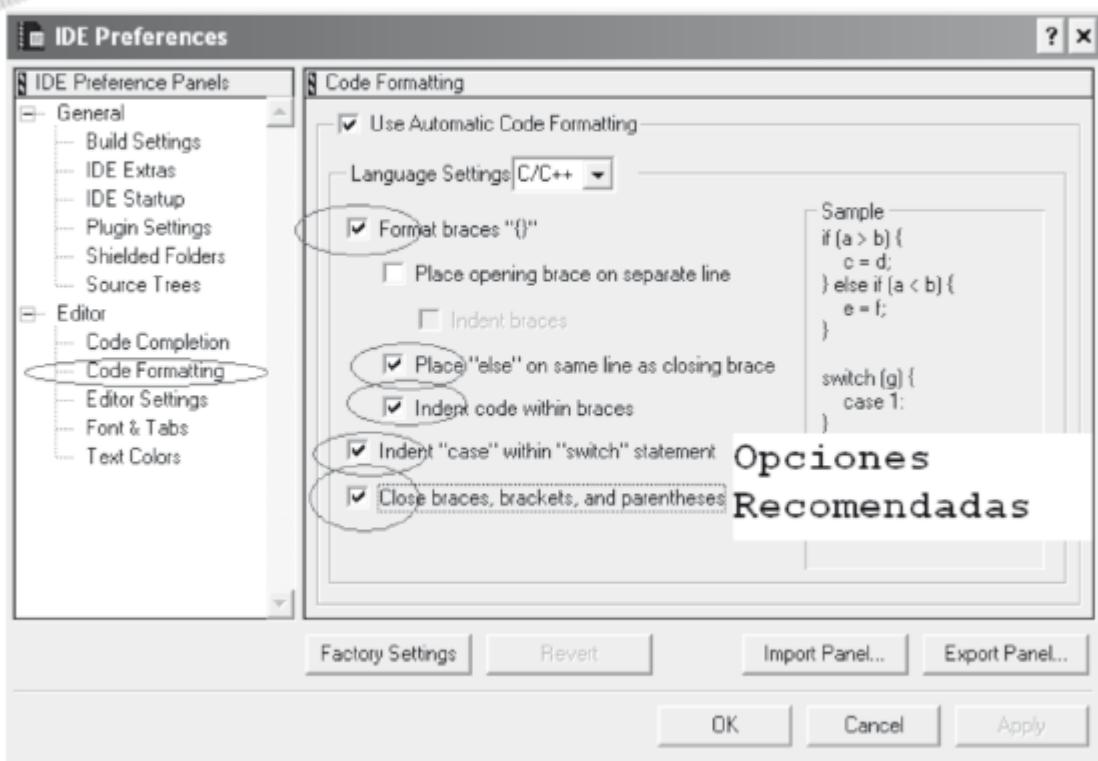
En la ventana **IDE Preferences → Editor Settings**, se pueden encontrar varias opciones que el programador puede modificar de acuerdo con sus gustos.

La opción recomendada es la mostrada en el *Gráfico 3.6*, la cual ayuda a que la edición sea más orientada al compilador, y a evitar que el programador olvide cerrar algún corchete o paréntesis y adicionar tabuladores en cada estructura nueva.

Entre los editores del mercado el Codewarrior® presenta un ambiente muy amigable y es bastante completo en su funcionalidad.

GRÁFICO
3.6

Opciones de compilador recomendadas de Codewarrior®.



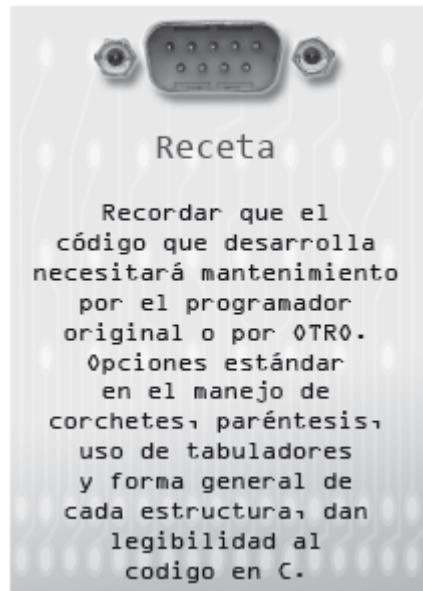
El Compilador de Codewarrior®

Es la parte del programa que realiza el proceso descrito en el Capítulo 1, mediante el cual los archivos del proyecto son llevados al lenguaje de máquina.

El Codewarrior® permite establecer varias configuraciones como el manejo de las optimizaciones a realizar, las cuales, dependiendo del diseñador, están orientadas a que el tiempo de ejecución sea lo más corto posible, o bien que el espacio ocupado en memoria sea el menor posible.

Existe una gran lista de opciones que el programador puede adicionar en la opción **Edit → Standard Settings → Compiler for HC08** en el botón “**Options**”, allí podrá seleccionar todas las que se consideren apropiadas según el proyecto, sea la prioridad la velocidad de ejecución o el espacio de memoria (*Gráfico 3.7*).

Una vez se presione el botón “**OK**” estas opciones regirán la compilación de este proyecto, si uno nuevo es creado, las opciones de optimización serán de nuevo las que trae el Codewarrior® por defecto.





Ventana de optimizaciones del compilador de Codewarrior®.



El optimizador del Codewarrior® le permite al programador establecer la configuración más adecuada para el proyecto, como un tiempo de ejecución lo más corto posible, o que el espacio ocupado en memoria sea el menor.

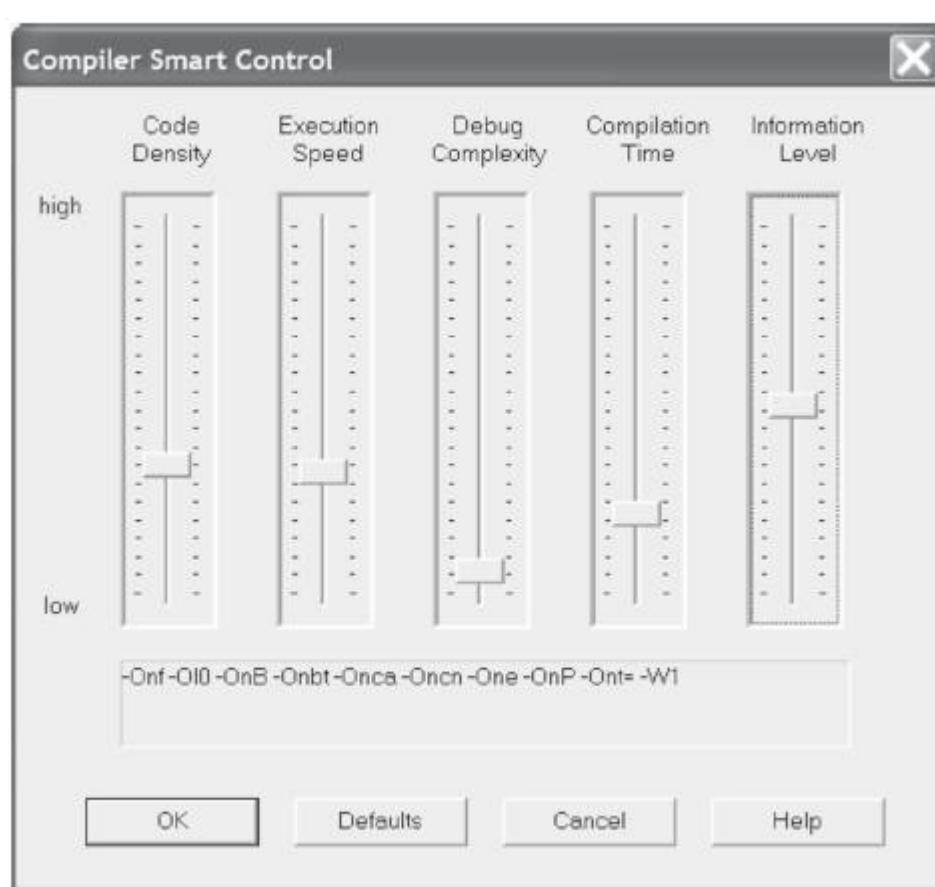
En cada una de las opciones de optimización el diseñador puede señalar y presionar el botón “Help” para una descripción detallada del efecto de dicha optimización.

En algunos casos el programador puede encontrar esta tarea un poco tediosa por la cantidad de opciones y el detalle de cada una. Estas opciones están disponibles, sin embargo hay varios detalles que deben tenerse en cuenta, como que el compilador está diseñado para optimizar el código generado, que se debe pensar que la ejecución del proyecto no puede ser tan sensible al tipo de optimización aplicado, y que la memoria es la suficiente para soportar el desempeño y la capacidad exigida por el proyecto.

En este caso y de forma práctica solo se recomienda el uso de la ventana **Edit → Standard Settings → Compiler for HC08** botón “Smart Sliders” (*Gráfico 3.8*) en la etapa final del diseño, cuando se quiere dejar el código lo más compacto posible.



Ventana de control de optimización.



Esta ventana permite de forma sencilla mover los cursores sobre la característica que se quiere enfatizar para la optimización de un proyecto, ya sea por espacio en memoria “**Code Density**” (low ocupará mayor espacio en memoria), o velocidad de ejecución “**Execution Speed**” (high generará código más rápido) principalmente.

Las demás opciones se refieren a características de depuración en las cuales el compilador puede generar un código en ensamblador mucho más sencillo “**Debug Complexity**”, o bien, en proyectos de gran tamaño donde el tiempo de compilación “**Compilation Time**” es apreciable, poderlo manejar a expensas de tener menor nivel de información “**Information Level**”.

La compilación de un proyecto una vez definidos estos parámetros se realiza mediante la opción **Project → Make** o de forma rápida mediante la presión de la tecla **F7**.



Para la optimización de código, usar los smart sliders, dependiendo de la optimización deseada de espacio o velocidad, en lugar de las opciones independientes en la ventana de optimizaciones.

El depurador de Codewarrior®

Un proyecto compilado satisfactoriamente, denota que no se tienen errores de sintaxis, entonces se procede a realizar la depuración del proyecto, el cual consiste en enviar el código de máquina al microcontrolador *target* y realizar su ejecución, sea de forma normal o con puntos de ruptura (*Break Points*), que permite conocer de cerca el comportamiento del programa diseñado.



Algunas zonas de programa resultan más sencillas de evaluar en modo “Full Chip Simulation”, que no requiere conexiones físicas. Especialmente secciones que no involucra el manejo de circuitos externos.

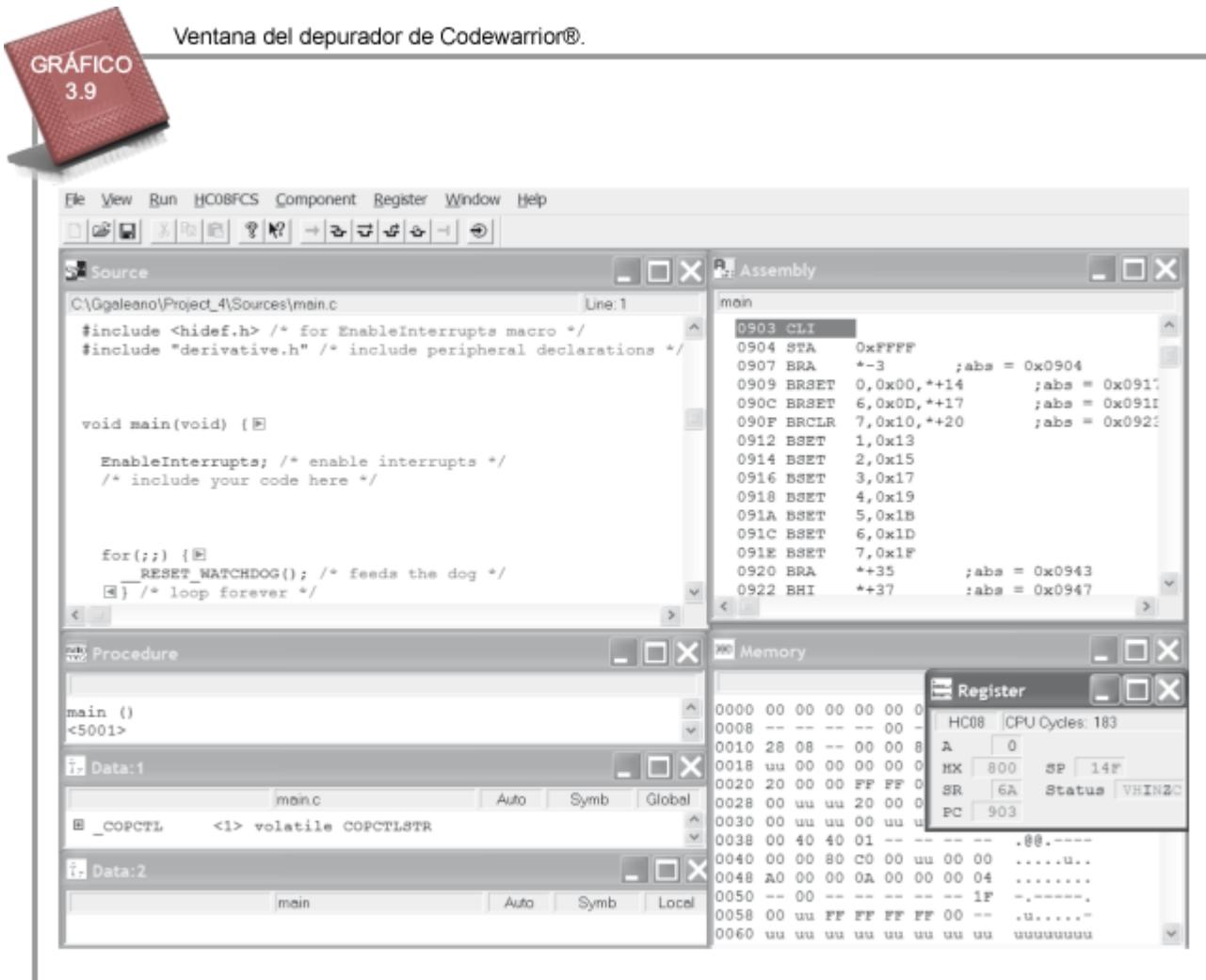
Para realizar la depuración se va al menú **Project → Debug** o bien de forma rápida y sencilla mediante la tecla **F5**.

Como se mencionó anteriormente en el paso 5, en la sección de creación de proyectos en C, se tienen dos tipos de depuración:

Full Chip Simulation: en la cual no se requiere del microcontrolador, porque toda la depuración se realiza en el mismo PC⁷ donde se realizó la compilación. Esta opción es muy cómoda debido a que no requiere de conexiones externas al PC o alimentación de tarjetas de evaluación y el tiempo de depuración es menor. Sin embargo, solo es recomendable para secciones de código en las cuales únicamente interfieren datos y no hardware externo.

Mon08 Interface / P&E Multilink: en el que el código de máquina es programado en el microcontrolador y su ejecución se realizará directamente en la máquina final.

Esta es la opción recomendada porque permite 100% interacción con el hardware, la simulación es completamente real y la lectura y escritura sobre los pines del microcontrolador se realizan de la misma forma como se ejecutará una vez el sistema de evaluación sea desconectado del PC.



Las subventanas del gráfico anterior pueden ser activadas o no, posicionadas y cambiadas de tamaño dependiendo de la etapa de depuración en la que se está, bien se puede querer mayor detalle del lenguaje C con la ventana “**Source**” y no tener las ventanas de ensamblador “**Assembly**” y de registros o modelo de programación “**Register**”, para fijar la atención en el código, en las funciones “**Procedure**” y en las variables globales “**Data:1**” y en las variables locales “**Data: 2**” o en otro momento, se puede requerir mayor detalle del lenguaje ensamblador por tratarse de una rutina crítica, la cual se quiere seguir paso por paso, instrucción por instrucción en avance del programa.



El depurador Codewarrior® permite al programador guardar configuraciones personalizadas, de modo que se pueden

La ventana de memoria “**Memory**” corresponde al mapa de memoria del microcontrolador, allí pueden ser visualizados de forma inmediata los registros internos del microcontrolador, los datos en RAM, la memoria de programa, los vectores de interrupción.

crear plantillas específicas para cada tipo de proyecto.

Una vez que cierta configuración ha sido personalizada ésta puede ser guardada **File** → **Save Configuration** y recuperada **File** → **Open Configuration** en otro proyecto. De esta forma se puede tener ventanas de depuración específicas para depuración en C, depuración en ensamblador, depuración de *buffers* de memoria etc.

Los botones de depuración

Una vez se ha posicionado en la ventana del depurador, se pueden realizar varias acciones, dependiendo del nivel de ejecución que se requiere, el sistema es sacado de *reset* o enviado a la función “*Startup*” donde realiza las inicializaciones del SP⁸ y de variables y de allí es enviando a la función main(), donde la ejecución es parada atendiendo a la instrucción del diseñador.

Dependiendo del nivel de detalle deseado se tiene acceso a los siguientes botones:



Run: realiza la ejecución desde la función main(), en el caso de una depuración “*In-Circuit*”, la ejecución en el microcontrolador se hará en tiempo real y de forma continua. El procesamiento puede ser detenido en cualquier momento mediante el botón “Halt”.



Step-In: realiza la ejecución de una línea simple de C, en el caso del llamado a una función, realiza su llamado y se posiciona en la primera línea de la función.

Es útil para evaluar el comportamiento de algún procedimiento de forma detallada.



Step-Over: realiza la ejecución de una línea de C; sin embargo, si la línea corresponde a un llamado de función, esta será invocada y su ejecución no será intervenida, se ejecutará en tiempo real y el PC⁹ detiene su ejecución al retorno de la función.



Step-Out: realiza ejecución hasta que el procesador abandone el procedimiento o función actual. Es útil cuando se entra a ejecución detallada *Step-In* de una función y se quiere realizar la ejecución de las demás líneas de la función y posicionarse en la línea siguiente después del retorno de la función.



Single-Step: ejecuta la instrucción en ensamblador que está siendo apuntada por el Contador de Programa “PC”. Útil cuando se requiere conocer el comportamiento del software a nivel de ensamblador y sus efectos sobre las variables, la memoria y los puertos de entrada y salida.



Halt: obliga al microcontrolador a detenerse en la posición en la que se encuentra el Contador de Programa “PC” y a actualizar las subventanas del depurador, esta función es solo habilitada en los procesadores HCS08, el procesador usado acá no lo soporta.



Reset: genera un *reset* del microcontrolador, obligando al contador de programa a posicionarse en la función de inicio “Startup” o bien la apuntada por su vector de *Reset* (0xFFFFE – 0xFFFF).

El código de arranque “Startup”

Una vez que el microcontrolador sale de *reset*, el contador de programa se carga con el valor que encuentre en las direcciones 0xFFFFE y 0xFFFF → (PCH:PCL).

Allí inicia la aplicación a ejecutarse antes de ir al main(){}, el sistema ejecuta un código previo que se encarga de realizar lo siguiente:

1. Inicializar el valor del apuntador de pila.

Al salir de *reset*, el SP¹⁰ se inicializa en 0x00FF, sin embargo, este valor no necesariamente corresponde al tope máximo de la RAM, por consiguiente deberá inicializarse en el valor apropiado.

2. Inicializar variables globales en cero

El estándar ANSI C define que las variables globales declaradas una vez se llegue al main tendrán el valor de cero. El “Startup” realiza un ciclo de borrado de la zona en RAM que realiza esta inicialización de forma rápida.

3. Inicializar variables con valores diferentes de cero

Esta sección se encarga de traer de la memoria flash los valores con los que inician ciertas variables que fueron cargadas con valores específicos antes de llegar al main(){}.



4. Invocar llamado al main(){}

Por cuestiones de velocidad de ejecución y de espacio ocupado en memoria de programa, el código previo “Startup” está desarrollado en lenguaje ensamblador. De forma opcional puede modificarse (solo para casos estrictamente necesarios), o inclusive solicitar que no sea incluido (excluir pasos 2 y 3), lo que permite que el arranque del sistema sea muy rápido (aunque no estándar C).

Al crear un proyecto en ANSI C, el Codewarrior® por defecto generará el código “Startup” adecuado, sin embargo la ventana de configuración en el paso 9 de creación de proyectos, pudiera modificarse para no incluir el código. Las

opciones mostradas en el paso 9 (ANSI startup code, small y none) son las recomendadas y el estándar de creación de proyectos estándar ANSI C.

7 PC: acá PC se refiere a computadora, no *program counter*.

8 SP: *Stack Pointer*.

9 PC: acá se refiere a *Program Counter* del microcontrolador.

10 SP: Stack pointer.

3.4 EL PRIMER PROGRAMA EMBEBIDO EN C USANDO CODEWARRIOR®

En este punto se puede empezar con el primer programa real que se ejecute en las tarjetas evaluación PIC-Link y AP-Link, el cual permitirá conocer el manejo de la tarjeta, familiarizarse mejor con el manejo del IDE del compilador y establecer el ciclo de diseño típico con el cual se realizarán todas las demostraciones en las demás prácticas que se desarrolle en el libro.

EJEMPLO No. 3

Encendido de led Freescale™ en C

Objetivo:

El objetivo del primer programa ANSI C es desarrollar un código que encienda el LED OUT-1 de prueba en el sistema de desarrollo AP-Link.

Solución:

El primer paso será entonces crear un nuevo proyecto que se llamará **LED_TEST**, se crea el proyecto siguiendo uno a uno los 10 pasos de creación de un proyecto nuevo, tal como se indicó en el numeral 3.3.1 de este capítulo, así:

1. New Project:
2. Definición de Core: seleccionar “HC08”.
3. Definición de subfamilia: seleccionar “AP Family”.
4. Definición de la parte específica: seleccionar “MC68HC908AP16A”.
5. Definir el tipo de conexión para simulación: seleccionar “Mon08 Interface”.

6. Nombre del proyecto: se digita el nombre “LED_TEST.mcp”
7. Add Additional Files: botón “Next”.
8. Uso de Processor Expert™: botón “Next”.
9. Opciones de C/C++: botón “Next”.
10. Uso del PC_Lint: botón “Finish”

Edición del programa:

Una vez creado el proyecto se abre el archivo main.c en el editor, y se posiciona en la función void main(void).

Ahora y según el diagrama de conexiones del sistema AP-Link, el LED de prueba 1, llamado VISUAL OUT-1 encenderá con un nivel “1” en el pin PTC3 del Puerto C del Microcontrolador AP16A, lo cual se hace con la línea:

```
PTC_PTC3 = 1; // poner un nivel “1” en el registro del Puerto C
```

Adicional a esta línea, se debe configurar este pin como pin de salida, lo cual se hace con un “1” en el bit 3 del registro de dirección DDR del puerto C, llamado

DDRC_DDRC3 así:

```
DDRC_DDRC 3 = 1; // configurar pin del puerto C como salida.
```

El código del primer programa se verá de la siguiente forma:

```
***** Ejemplo 3 *****

// Encendido de Led Freescale en C

// Fecha: Feb 4,2009

// Asunto: Encendido LED OUT-1 usando Codewarrior®

// Hardware: Sistema de desarrollo AP-Link(2008-01-14)

// para Microcontrolador Freescale AP16.

// Version: 1.0 Por: Gustavo A. Galeano A.

*****
```

```
#include <hidef.h> /* for EnableInterrupts macro */
```

```

#include "derivative.h" /* include peripheral declarations */

void main(void) {

    EnableInterrupts; /* enable interrupts */

    /* include your code here */

    PTC_PTC3 = 1;

    DDRC_DDRC3 = 1;

    for(;;) { __RESET_WATCHDOG(); /* feeds the dog */

    } /* loop forever */

    /* please make sure that you never leave main */
}

```

Se finaliza la edición del programa.

Compilación del programa:

La compilación se realiza presionando la tecla **F7**.

El compilador no deberá mostrar errores de sintaxis, de no ser así se debe verificar que el código esta bien digitado comparándolo con el código mostrado.

Depuración del programa:

Realizar ahora la conexión del sistema AP-Link por el puerto serial y alimentar la tarjeta, verificar que el Jumper JP1 está en la posición **DEBUG** (ver Gráfico 1.29 en el capítulo 1)

Luego se programa y realiza la depuración sobre el microcontrolador, para esto se presiona la tecla **F5**, la cual ejecutará una nueva aplicación que intenta conexión con el sistema AP-Link; recordar que la conexión se realiza a 9600 Baudios, asegurándose de seleccionar el puerto (COM) por el cual esta conectado el sistema AP-Link y que la ventana de programación tiene la selección Class 1.

Para ejecutar el programa en memoria del microcontrolador presione el botón  o bien la tecla **F5**.

Con esto, el LED de prueba OUT-1, deberá permanecer encendido.

De no ser de así, intente generando reset con la presión del icono  o de la tecla **F6**

Ejecutar el programa paso por paso con la presión del icono  o la tecla **F10**.

Una vez el cursor esté en la línea **DDRC_DDRC 3 = 1;** deberá encender el LED.

Discusión:

El programa realizado fue editado y compilado, y quedará residente en la memoria flash del microcontrolador, que se ejecutará automáticamente si el microcontrolador es energizado y sin dependencia del PC.

Para evaluar esta operación, se debe desconectar el cable serial DB9 del PC y cambiar el Jumper a la posición RUN y el LED deberá encenderse y permanecer así, indicando que el programa está siendo ejecutado.

RESUMEN DEL CAPÍTULO

A la hora de seleccionar un software de desarrollo en C, se deben considerar varios aspectos sobre el fabricante, su calidad y trayectoria, así se asegura que la inversión es la apropiada y facilitará el proceso de diseño.

Desde el punto de vista de la herramienta, son importantes los aspectos de integración de los tres componentes: Editor, Compilador y Depurador. El soporte técnico y la vigencia de las licencias, su portabilidad y posibilidad de actualización a otras plataformas.

Los ejemplos realizados en este texto, son desarrollados en el ambiente de Codewarrior® y CS, que permiten, la creación de un proyecto nuevo, su posterior compilación y depuración, garantizan el funcionamiento tal y como es especificado.

Establecer opciones estándar en la edición de un programa, permite tener un código sencillo y de fácil mantenimiento. Se recomienda el buen manejo de los corchetes, los paréntesis y los tabuladores en las diferentes estructuras que ayudan a la legibilidad del código.

El ciclo de diseño del programador, está basado en prueba y error del código compilado, y es acá donde se invierte la mayor parte del tiempo de desarrollo, de ahí la importancia de ubicar en cada compilador los menús de navegación y teclas rápidas que permitan que la evaluación del código se verifique de forma eficiente.

PREGUNTAS Y EJEMPLOS SUGERIDOS

Enumere los componentes básicos de un IDE.

¿Cuáles son las labores del código de arranque Startup, en un proyecto hecho en C?

¿Por qué es necesario optimizar el uso de la zona de RAM directa?

¿En qué casos se requiere optimizar la velocidad de ejecución de un programa? y ¿en cuáles es preferible optimizar la capacidad de memoria usada?

¿Qué utilidad se puede encontrar en optimizar el tiempo de compilación? ¿En qué caso sería útil tener un tiempo de compilación corto?

¿Qué procedimientos realiza la función *Startup* al arrancar un sistema embebido? ¿Cómo pudiera mejorarse el tiempo de ejecución del *Startup*, en sistemas que requieren un arranque muy rápido?

INTRODUCCIÓN

Este capítulo muestra de forma detallada las ventajas que ofrece el lenguaje de alto nivel C, frente a la programación en lenguaje de ensamblador, estas razones son importantes para justificar el aprendizaje de este lenguaje de programación que traerá beneficios al diseñador de sistemas embebidos.

Históricamente se puede recordar la forma como se iniciaba la programación de microprocesadores en instituciones: se tenía que lidiar bastante con la conexión hardware-software y entender que no siempre lo que se escribe y se edita es lo que realmente hace “funcionar” el hardware.

La experiencia como programadores ha enseñado que gran parte del tiempo de desarrollo se invierte en buscar las razones por las cuales un circuito no obedece a lo que aparentemente el software ordena. Este ciclo de verificación del software, simulación, programado del microcontrolador, depuración y prueba del sistema es repetido de manera rutinaria hasta lograr encontrar los “bugs” del software o mal funcionamiento del hardware y llevar todo el sistema a su completa especificación.

El capítulo muestra como el lenguaje C ofrece una herramienta de programación eficaz para que el proceso de diseño sea más pragmático, sin limitar tampoco al diseñador en su filosofía propia de programación; sin embargo, se deben tener en cuenta algunas consideraciones y limitaciones importantes antes de establecerlo como herramienta única de desarrollo. Se deben aprovechar de sus ventajas, pero también conocer las limitaciones que se tienen para poder minimizar su efecto en cada línea de programa.

En la segunda parte se muestran las diferencias básicas entre la programación para computadoras comerciales y la programación con microcontroladores; estas consideraciones están dirigidas a los programadores de alto nivel, quienes deberán tener presente que las pequeñas computadoras cuentan con limitaciones especiales para su programación.

El resto del capítulo enseña al programador como realizar la declaración de constantes y de variables en memoria RAM con sus respectivos modificadores.

4.1 VENTAJAS Y DESVENTAJAS DEL C USADO EN SISTEMAS EMBEBIDOS

Ventajas:



El lenguaje C es una herramienta de programación altamente estandarizada y muy popular entre los diseñadores de software, lo que constituye una ventaja,

a. El lenguaje C es un lenguaje universal

Mientras el lenguaje ensamblador es exclusivo para determinada máquina, marca y arquitectura, el lenguaje C es un lenguaje universal y ampliamente estandarizado. Desde el momento de su creación por los ingenieros Ritchie y Kernighan su crecimiento y popularidad han superado las expectativas, y más ahora que entra en el mercado de los procesadores embebidos su crecimiento es aún mayor.

Este hecho es tal vez el más importante de resaltar, ya que garantiza el encontrar con facilidad programadores en el medio que puedan digitar una aplicación, y no estar limitados a los expertos que solo conocen determinado lenguaje ensamblador y/o arquitectura.

b. Portabilidad del código

Siendo el C un lenguaje tan popularizado, la creación de compiladores para cualquier máquina ha sido más que una necesidad, un deber de cualquier fabricante de procesadores, garantizando que cada línea de código que se escriba no sea exclusiva, o solo se ejecute en determinado procesador.

Considerar el hecho que la aplicación tenga la posibilidad de ser ejecutada en todas las marcas y arquitecturas posibles, garantizará la continuidad del negocio y de la aplicación particular, independiente de la permanencia de determinado procesador. Existe una ventaja implícita en esta opción y es el hecho de poder simular código en una máquina diferente a la que va a ejecutar el sistema embebido; en algunas ocasiones, y especialmente con aquel código que poco o nada tiene que ver con el hardware, es preferible simularlo en una máquina con mayores y mejores recursos de simulación, que tomar el sistema embebido, hacer la programación solo para evaluar código exclusivamente lógico. Una vez el código está ejecutándose en la máquina inicialmente seleccionada, es fácil migrar hacia una nueva arquitectura, aprovechando no menos del 95% del código realizado (...y probado).



La portabilidad inherente al lenguaje C hace posible probar las aplicaciones en diferentes máquinas, lo cual ahorra tiempo porque se puede simular el funcionamiento del software en equipos de mayor potencia al que va a ejecutar el

ya que facilita su implementación en múltiples equipos



sistema embebido.

Programar en lenguaje ensamblador es dispendioso y hace difíciles los cambios posteriores; en cambio, las secuencias de control propias del lenguaje C pueden ser sin problema por diseñadores que se sumen al proyecto más adelante.



En la realidad se tienen proyectos que han iniciado en procesadores de 8 bits y que luego, por algún requerimiento del mercado, limitaciones de consumo de energía, disminución de precios o actualización de tecnología, deben soportarse en arquitecturas superiores de 16 bits (ejemplo Freescale™ HC12 o HCS12) o superiores, donde el tiempo de desarrollo y la facilidad de rodar la aplicación en la nueva arquitectura ha sido el factor fundamental, y de no ser porque el código ha sido elaborado en C, se tendría que haber iniciado la aplicación desde el principio y nuevamente probar todo el código.

c. El lenguaje C facilita el trabajo en equipo

Esta es una gran ventaja respecto a la programación en lenguaje de ensamblador en la cual el trabajo en equipo es complejo.

La estructura del lenguaje C que usa secuencias de control, resulta muy útil a la hora de enfrentar un proyecto en un equipo de trabajo de cualquier tamaño, en el cual se puede dividir el proyecto en módulos que pueden a su vez ser asignados a programadores diferentes.

En el caso de una compañía de investigación y desarrollo resulta muy útil incorporar al grupo de trabajo varios programadores para disminuir el tiempo de diseño. Resulta muy apropiado para las compañías tener un código desarrollado en alto nivel que un nuevo programador pueda retomar, hacerle las modificaciones que pide su cliente, agregarle o quitarle código de manera rápida y sencilla sin afectar la estructura general del programa y sin dañar lo que ya ha sido ensayado y funciona.

Proyectos desarrollados en lenguaje ensamblador han mostrado experiencias no muy buenas, porque al retomar el código para hacerle alguna modificación o mejora, se ha tardado más tiempo tratando de entender cómo se diseñaron las rutinas, que en realizar la modificación en sí. En el mejor de los casos la inversión es solo de tiempo, sin mencionar que posiblemente estos cambios afecten procedimientos que ya funcionaban o estaban optimizados.



Todo proyecto
requerirá, tarde
o temprano, algunos
cambios, bien sea para
adaptarse a nuevas
tecnologías o por
solicitud del cliente. Los

Los cambios siempre serán necesarios, los clientes piden modificaciones, sugieren mejoras, algunos desean comportamientos muy particulares, los cuales, en algunas ocasiones, no son de fácil implementación; la dirección de mercadeo de cada compañía exige nuevas y mejores versiones. No existirá entonces en el *firmware* una versión definitiva y es labor del programador garantizar, además del correcto funcionamiento del prototipo, que el código es lo suficientemente flexible a cambios, adiciones y mejoras.

Los códigos desarrollados en lenguaje C permiten que el soporte por fabricantes y personas externas a la compañía pueda hacerse con más facilidad; en muchas situaciones se tienen programas desarrollados en lenguaje ensamblador bastante complicados porque se ha querido economizar código y RAM, de modo que resulta muy difícil ayudarlos y se tiene que iniciar el proyecto de nuevo (en C o en ensamblador), en lugar de tratar de modificar el código. Hasta este punto se ha considerado que los cambios se han solicitado al programador original del código, pero si se trata de un nuevo programador que necesite retomar el código la situación se vuelve aún peor.

En casos donde por alguna razón se ha tenido que modificar un código en ensamblador se ha requerido el tiempo para “entender” lo que el programador original digitó, comprender su filosofía de programación y tener el “set” de instrucciones y sus diferentes modos de direccionamiento a mano, para después de eso entrar con alguna confianza a realizar la modificación. En muchos casos se ha optado mejor por tomar las especificaciones iniciales del proyecto y reescribir todo el código de nuevo. Para estos casos (y sobra decirlo), es mucho mejor dirigir el problema al diseñador original, sin embargo, en algunos casos no es posible, ya que de la misma manera que los programas, las personas son dinámicas, en unas ocasiones hacen parte de cierta compañía como diseñador pero luego pueden no estar o estar en otra área y poco les interesaría revisar programas, o mucho peor, estar en otra compañía donde ya no sea posible tenerlos en frente del código.

códigos desarrollados en C permiten que muchas de estas modificaciones provengan del soporte dado por fabricantes y programadores externos.

El desarrollo de un software embebido siempre será propiedad de la compañía que lo desarrolló, o en su defecto, del cliente final. Por ello es esencial que el código utilice un lenguaje de alto nivel y que sea bastante estandarizado, propiedades atribuibles a C.

En fin, desarrollando en lenguaje de alto nivel, el código es para la compañía y no para el programador, es más para el equipo y no para un individuo particular, y cuando de trabajo en equipo se trata se puede compartir código, integrar nuevos programadores al proyecto y retomar el código desarrollado por otros.

d. El lenguaje C permite la modularización

“Divide y vencerás” dijo Napoleón, y aplica tanto para encontrar problemas de hardware como de software cuando están claramente divididos.

Si bien es cierto que es posible modularizar programas en lenguaje ensamblador, no es la tendencia actual de los programadores. En muchos casos, por estar sumergidos en el código, se olvida documentar bloques importantes, crear nuevos archivos con el código correspondiente a un bloque de software resultando que el programa inicial, que en un solo archivo en ensamblador (.ASM) parece ser pequeño, empieza a crecer a medida que más cambios y nuevas opciones se agregan, y luego por el temor a que modularizando el código final no funcione, se deja tal cual.

El objetivo es proveer al programador de una nueva técnica, por eso es importante desde ahora involucrar el tema de la modularización, de manera similar a como se concibe en el hardware los módulos. Se tienen así módulos de comunicación (RS-232, CAN, USB, GPRS, RS-485, LIN...), módulos de visualización (Leds, 7 Segmentos, LCD texto, LCD gráfico), módulos de entrada (botones, teclados, sensores), módulos que han permitido encontrar problemas de hardware de manera sencilla, hacer cambios en uno de sus módulos (sin afectar el resto) y tener diferentes versiones de un hardware con bloques comunes.

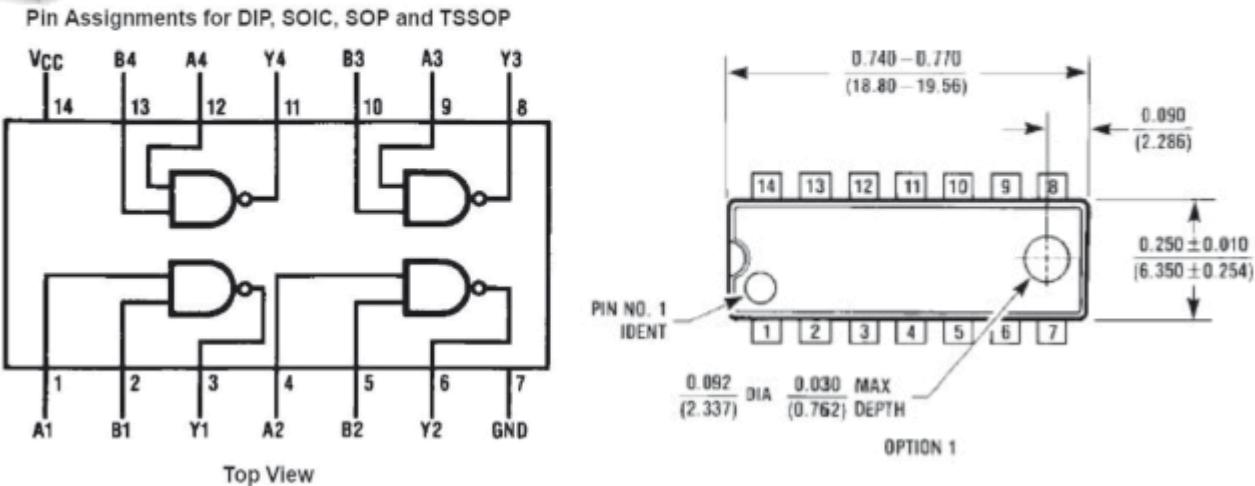
e. El lenguaje C permite el encapsulado

Una de las mayores ventajas de C es que posibilita trabajar por módulos, es decir, un proyecto será escalable según se presenten nuevas necesidades.

El concepto de encapsulado es bien claro cuando se habla de un circuito integrado, y para el caso considere la compuerta básica NAND National MM74HC00 (*ver Gráfico 4.1*). Para usar las especificaciones del circuito integrado es necesario proporcionarle la alimentación por los pines 7 y 14, una vez se haga esto por los pines Ax y Bx entran 2 valores digitales que se quiere procesar, el resultado o salida se obtiene por el pin Yx correspondiente, ambas señales (las de entrada y las de salida) están bien especificadas, sus niveles máximos y mínimos de voltaje que aceptan las entradas y los resultados que se obtienen a la salida.



Compuerta 74HC00 NAND y su encapsulado¹.



El encapsulado propio del integrado solo permite interactuar con los niveles del exterior (alimentación, Ax, Bx, Yx), mas no permite hacerlo con las señales intermedias o internas.

El concepto de encapsulado en software es de los más importantes en la programación orientada a objetos (POO), porque permite tener bloques y módulos de programas específicos con entradas llamadas argumentos, y salidas denominadas retornos, ambos muy bien dimensionados.

El encapsulado de software permite realizar procesos muy específicos y robustos, ya que el programador puede proteger por medio del encapsulado, tanto como lo considere necesario, las variables y procedimientos, haciendo que estos no puedan ser accesados o modificados externamente, o si lo hacen, sea a través de un procedimiento o función pública diseñada específicamente para este fin.

Si en el momento se siente que esta es una limitación en realidad no lo es, se trata de proteger los procedimientos y variables de accesos no permitidos o de cambios involuntarios que hagan colapsar la aplicación y se desconozca el módulo que pudiera causar el error.

El concepto de encapsulado es uno de los más importantes en la programación orientada a objetos (POO), porque permite tener bloques y módulos de programas específicos con entradas llamadas argumentos, y salidas denominadas retornos.



Con C el programador podrá realizar una prueba del prototipo mucho más rápido que si trabajara con el lenguaje máquina. Las pruebas iniciales ayudan a enfocar el diseño y a detectar aquellos problemas que solo se aprecian cuando el sistema se encuentra en funcionamiento.

f. El Lenguaje C es una herramienta rápida de programación

Para el caso de los sistemas embebidos resulta una herramienta bastante eficiente en tiempo de desarrollo; el tiempo que tarda en realizarse un prototipo funcional y de allí a llevarlo a la etapa de producción, es un aspecto muy importante en términos económicos: menor tiempo de desarrollo significa menor inversión en desarrollo a un producto; por lo mismo, un tiempo de desarrollo corto puede hacer viable el proyecto, porque permite recuperar económicamente la inversión en menor tiempo y con menos unidades vendidas, significa tener el producto en producción en un tiempo menor y con esto ganar tiempo de manufactura, lo que conlleva a estar en el mercado antes que la competencia e identificar y corregir problemas del sistema embebido con mayor rapidez (*ver Gráfico 4.2 y 4.3*).

El equipo en sí contará con mayor tiempo en el campo, lo cual permite su óptima maduración y obtener retroalimentación más rápida sobre el comportamiento del mismo. De forma similar los cambios se pueden llevar a cabo de manera rápida con lo que el tiempo de respuesta a los cambios que solicita el mercado son mucho más cortos, permitiendo de esta forma obtener utilidades mayores y rápidas para la compañía desarrolladora.

GRÁFICO
4.2

Cronograma de diseño en lenguaje ensamblador (1 ingeniero de software).

ACTIVIDAD	24/4/07	6/5/07	13/5/07	20/5/07	27/5/07	3/6/07	10/6/07	17/6/07	24/6/07	1/7/07	8/7/07	15/7/07	22/7/07	29/7/07	5/8/07	12/8/07
DEFINICIÓN																
PRUEBAS INICIALES																
HARDWARE																
MUESTRAS																
ESQUEMÁTICO																
PCB																
FABRICACIÓN																
PROTOS																
SOFTWARE																
PRUEBAS CAMPO																
INICIO PRODUCCIÓN																



**GRÁFICO
4.3**

Menor TTM² significa menor inversión y mayores utilidades.



CRONOGRAMA DE ACTIVIDADES TÍPICO DE UN DISEÑO												
ACTIVIDAD	29/4/01	6/5/01	13/5/01	20/5/01	27/5/01	3/6/01	10/6/01	17/6/01	24/6/01	29/7/01	5/8/01	12/8/01
DEFINICIÓN												
PRUEBAS INICIALES												
HARDWARE												
MUESTRAS												
ESQUEMÁTICO												
PCB												
FABRICACIÓN												
PROTOS												
SOFTWARE												
PRUEBAS CAMPO												
INICIO PRODUCCIÓN												



Una característica del lenguaje C es que admite incorporar código ensamblador en las partes en donde el programador lo considere necesario.

Las estructuras y palabras reservadas del C son mucho más fáciles de memorizar que las instrucciones y modos de direccionamiento del microcontrolador para el cual se programe.

g. El lenguaje C permite la incorporación de bloques de ensamblador



Teniendo como plataforma de desarrollo el ANSI C, es permitida la incorporación de código ensamblador donde se considere necesario, cosa que no es posible en ensamblador.

Bastará con abrir un solo “{” corchete e incluir todo el código en ensamblador propio de la máquina y suspender la incorporación del mismo cerrando el corchete “}”, el código contenido es respetado al momento de generar el lenguaje de máquina garantizando así el deseo del programador.

Esta ventaja es bien valorada por los programadores expertos en lenguaje ensamblador, debido a que el C no les limita la posibilidad de adicionar cortos códigos en ensamblador donde lo considere apropiado, también permite adicionar rutinas ya probadas y de confianza del programador e interactuar con las funciones y variables de alto nivel, así como llegar al hardware de una manera muy directa, y de acuerdo al criterio del diseñador, reemplazar código generado por el compilador por el propio.

h. El compilador usa todo el set de instrucciones

Como se ha dicho, el compilador a su vez es un programa de software, al cual se le indicaron como usar cada una de las instrucciones del microcontrolador elegido. Tiene la ventaja que no olvida nunca usar alguna y conoce bastante bien cada modo de direccionamiento, cosa que no pasa con el programador en ensamblador; en muchas ocasiones, por facilidad o por nivel de complejidad de algunas instrucciones, limita su uso.

Desde el punto de vista del programador resulta mucho más fácil memorizar todas las estructuras y palabras reservadas del C que todo el set de instrucciones y sus modos de direccionamiento de un microcontrolador en particular.



Proyecto en C vs. proyecto en ensamblador.

```
/*main.c*/
void main(void) {
    Mcu_Init();
    RTI1_Init();
    TimerInit();

    EnableInterrupts;

    GPS_Config();
    GPS_Init();

    for(;;){
        GPS_Process();
    }
}
```

```
; main.asm
    BSR  Mcu_Init
    JSR  RTI1_Init
    JSR  TimerInit
    CLI
    JSR  GPS_Config
    JSR  GPS_Init
LF :   JSR  GPS_Process
        BRA  LF
```

Desventajas

a. El lenguaje C ocupa mayor espacio en memoria de programa

Esta desventaja es válida, sin embargo se tendría que comparar la función del compilador con un excelente programador en ensamblador, que conozca muy bien y use el 100% de las instrucciones y modos de direccionamiento de procesador.

El código en C compilado a lenguaje de máquina ocupará más espacio en memoria de programa (memoria Flash), es por esta razón que el compilador no es recomendado para algunos proyectos donde el espacio en memoria de programa es muy crítico o el costo del procesador seleccionado se vea afectado por el hecho de usar un compilador; en aplicaciones sencillas el lenguaje C se vuelve inefficiente y genera código redundante que ocupa más espacio y tarda un mayor tiempo en ejecutarse.



En los proyectos donde el tamaño y la velocidad del programa sean críticos porque el procesador es de baja capacidad, el lenguaje ensamblador presenta ventajas frente al C; pero el

De tal manera que nunca se debe dejar de seguir considerando el lenguaje ensamblador cuando la aplicación embebida requiere aprovechar al máximo la velocidad de ejecución y/o ahorrar memoria tanto de programa como RAM.

b. Los buenos compiladores tienen un costo, los ensambladores no

Para todas las arquitecturas el ensamblador es gratuito, inclusive algunos macro-ensambladores incluyen un excelente ambiente de desarrollo, con simulador y programador de memoria flash. Para el caso de los compiladores de lenguaje C se debe considerar el costo asociado, el cual en algunos casos, se emite como licencias por cada máquina en la cual es instalado, dependiendo también de la capacidad en bytes a generar, número de archivos en el proyecto, etc. Esta limitación no se tiene cuando se habla de ensambladores.

La tendencia de los compiladores de procesadores de 8 y 16 bits es a proveer licencias de uso libre para una capacidad determinada de código, pero mientras la capacidad sea limitada en este aspecto el ensamblador tiene esta gran ventaja.

c. Los compiladores son a su vez programas y pueden tener “bugs”

Es por esta razón que pueden tener errores en sus procedimientos de optimizado al momento de generar el código de máquina.

Existe en el mercado una gran cantidad de fabricantes de compiladores para microcontroladores, cuya única diferencia aparente es el precio; sin embargo, detrás del precio no solo se está pagando por el compilador en sí sino por la experiencia, el respaldo, el servicio y el soporte.

Esta situación inclina un poco la balanza de las desventajas a favor de ensamblador, donde cualquier error de funcionamiento el programador lo buscará en el código que hizo, en su hardware, mientras que al usar un compilador de procedencia un poco cuestionable, deberá preocuparse también si el código generado estará bien convertido y si el optimizador no pasó por alto alguna situación de especial interés para el programador.

Por fortuna, los compiladores ya están bastante depurados en especial para los fabricantes más importantes; sin embargo, al momento de usar un compilador lo mejor será asegurarse del tiempo que la compañía lleva en el mercado para no llevarse sorpresas una vez se esté involucrado en un proyecto. Esta situación nunca se ha tenido en cuenta, debido a que no ha sido necesario cuando se considera la programación en ensamblador.

Es importante entonces resaltar el hecho que los compiladores son programas desarrollados por programadores como el lector mismo y que como tal pueden tener revisiones, por ello siempre es mejor contar con la última versión del compilador. En este aspecto es más confiable el ensamblador, su

avance tecnológico ha hecho que cada día se mejore la capacidad de memoria y la velocidad de los microcontroladores.

Detrás del precio de un compilador está la experiencia del fabricante, el respaldo ofrecido, el servicio y el soporte. Usar un compilador de procedencia cuestionable puede generar algunas inconsistencias en el código generado y ser causa de complicaciones en el proyecto más que una solución adecuada.

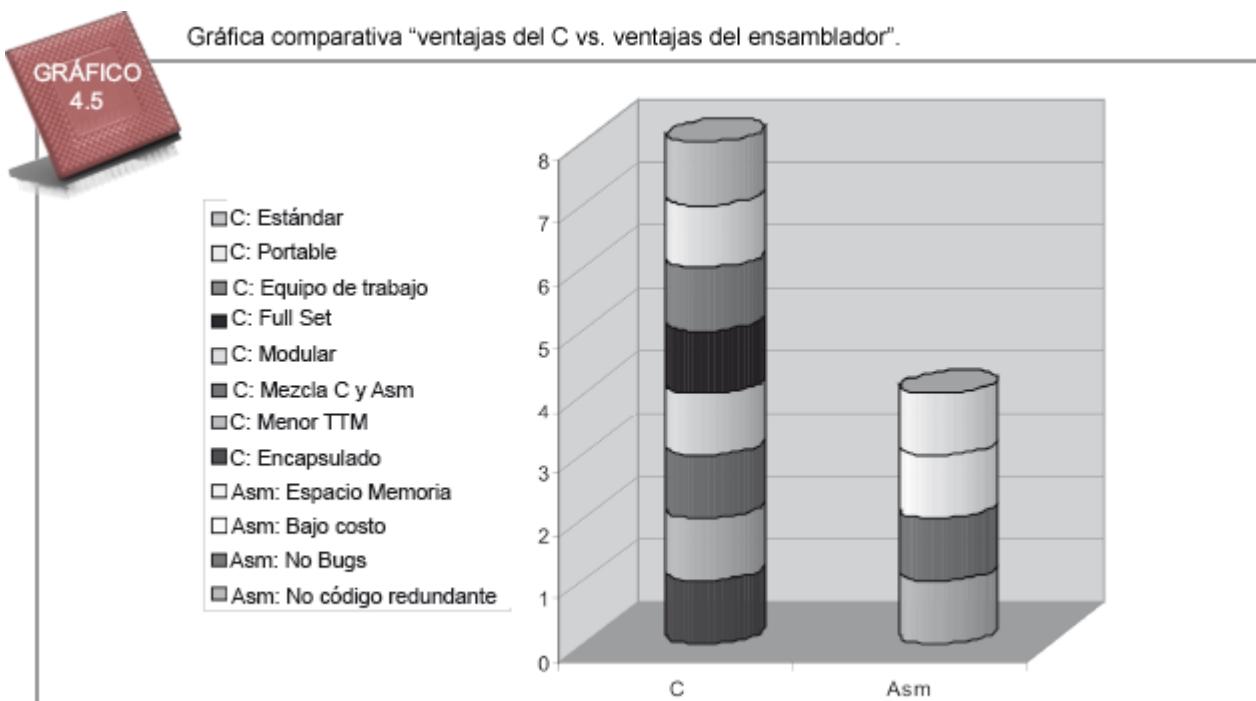


diseño es mucho más sencillo y directo, no tiene procesos de optimización, el ensamblador solo pasa cadena de caracteres (mnemónicos) a su respectivo código equivalente en lenguaje de máquina de manera segura.



d. Los compiladores generan código innecesario y redundante

Código adicional que el procesador posiblemente nunca ejecute, código que nunca se agregaría si el código se hiciera en ensamblador. Este código, además de ocupar un espacio adicional en la memoria flash, consume tiempo de ejecución si es procesado. En aplicaciones críticas de consumo de energía ejecutar mas instrucciones implica mayor corriente, cuando en aplicaciones en ensamblador este código no existiría y el procesador estaría mayor tiempo en modo de bajo consumo *wait, stop*, este último aspecto se ve aun más afectado cuando se incrementan las instrucciones adicionales generadas. Ver gráfico comparativo de ventajas vs. desventajas del lenguaje C en el *Gráfico 4.5*.



¹ Gráfico cortesía de National Semiconductor.

² TTM: Time To Market, tiempo transcurrido desde la concepción del producto, hasta tener la

primera unidad funcional en la línea de producción.

4.2 PROGRAMACIÓN EMBEBIDA CS. PROGRAMACIÓN PARA PC

El ambiente de programación de un sistema embebido es muy diferente a una computadora normal. Cada instrucción del programa debe recordar que tipo de procesador se está programando y sus limitaciones. Dada la orientación de este libro, se está considerando que las máquinas que ejecutarán el código final serán procesadores de 8 bits, con velocidades de ejecución máxima de 25 MHz, con limitación tanto de memoria de programa (Flash) como de datos (RAM).

La aclaración anterior es aún más importante para los programadores de sistemas o programadores de computadores o desktops, donde no existe una limitación aparente en la velocidad de ejecución ya que se está por encima de los 400MHz, tampoco existe preocupación por el tamaño del archivo ejecutable (.exe), o si la aplicación consume poca o mucha memoria RAM del sistema. Es de suponer que para este tipo de programadores estos tres aspectos son ilimitados y poco afectan en el desempeño final de la aplicación.

En el mundo embebido, los tres aspectos anteriores son de suma importancia, en muchas de las aplicaciones a cambio de disminuir el consumo de energía se requiere también disminuir la velocidad de ejecución a velocidades por debajo de 1MHz; es muy típico este cambio de velocidad, en especial cuando el sistema esta soportado por baterías, y por cuestiones de costo seleccionar procesadores con poca capacidad de memoria flash, por ejemplo 1024 bytes, y 128 bytes de RAM (en la cual estará el stack, las variables locales, globales y estáticas, conceptos que se aclararán en capítulos posteriores).

Supóngase que 28 bytes de RAM se dejan para todo el movimiento del stack (lo cual es bien limitado), se tendrían 100 bytes de datos, de allí, que cada declaración de una variable entera (2 bytes) equivale al 2% del total de la memoria, cifra muy diferente cuando se habla de un computador personal que tiene un valor típico de 512 Megabytes de RAM, donde la misma variable equivale al 0.00000039 % del total de la memoria.

Esto considerando solo la memoria RAM, en el aspecto de la memoria de almacenamiento de programa las distancias son aún mayores, se estaría comparando un disco duro de 60.000.000.000 (60 GBytes), con un pequeño bloque de almacenamiento de 1024 bytes, para el programador, usando la gran máquina, no importa si su archivo final .exe ocupara 100KBytes, 1 MBytes o 10 MBytes de memoria, mientras que para los programadores embebidos será imperativo conservar cada byte de memoria, de esta forma podrá hacer que su programa no se pase de una capacidad establecida y conservar los límites de precio acordados de la aplicación.

4.3 CONSTANTES Y VARIABLES



La principal diferencia entre programar para un sistema embebido y para un PC es la limitada capacidad en memoria y procesamiento de los sistemas embebidos, que obliga al programador a sacar el máximo provecho del equipo por medio del código que diseñe.





En el sistema embebido la definición de las constantes y la variables de manera adecuada es uno de los aspectos más importantes a considerar. La declaración mas pequeña en medida de bytes, generará al momento de la compilación un código mas resumido tanto en espacio de memoria como en la velocidad a la cual la procesa.

4.3.1 Constantes

Las constantes son valores que el compilador reemplazará en su primera pasada, las cuales no sufrirán ninguna modificación durante su ejecución. Estas constantes estarán siempre en la memoria de programa, y dependiendo de su definición podrían estar ubicadas en el transcurso del programa o en una ubicación fija en la memoria; para el primer caso, se hace referencia a definiciones mediante la directiva de compilación `#define`, que es equivalente a la muy familiar directiva de ensamblador “`EQU`”.

En realidad el `#define` se puede usar para muchos tipos de igualdades, en todos los casos facilitando entender el programa y futuras modificaciones. Estos simplifican mucho el hecho de modificar un parámetro con solo una alteración en una línea con efecto en cada línea que use la constante, además de simplificar evitan también daños inesperados en el código víctimas de un cambio, los cuales siempre serán pedidos por la aplicación.

Las constantes pueden ser de cualquier tipo de datos y además pueden tener operaciones para resolver el valor final de la constante, que sería equivalente a tomar una calculadora, realizar la operación y hacer un `#define` equivalente al valor final de la constante; en esta situación resulta mejor el primer caso porque es mucho más ilustrativo y se conoce el origen de la misma.

A continuación se verán varios ejemplos de declaración de constantes y su uso, también serán discutidos algunos casos en los cuales resulta de gran utilidad su declaración.

Declaración de las constantes

Para la declaración de las constantes se tienen a disposición dos formas dependiendo de la naturaleza de la misma:



La directiva de compilación `#define` asigna un valor final a constantes de cualquier tipo, su uso adecuado posibilita al diseñador el realizar cambios en el transcurso del programa alterando una sola línea de código, porque el `#define` automáticamente afectará a todas las líneas que la contengan.



El primer caso en el cual solo se desea que el valor sea reemplazado a lo largo del código se usa la sentencia #define con el signo '#' en la primera columna del editor. Algunos ejemplos:

```
#define FIN_CADENA      '/r' /*constante <enter>*/  
  
#define NRO_PI          3.1415926536  
  
#define TRUE            1  
  
#define FALSE           0  
  
#define AREA_CIRCUNFERENCIA NRO_PI*RADIO_MAX*RADIO_MAX
```

El segundo tipo utiliza la secuencia 'const'³ antes de la declaración de la constante, en este caso la constante se refiere a aquellas que por la naturaleza deben ocupar un lugar en la memoria flash, y aunque su valor no se puede cambiar (por ser constante), el código deberá acceder a una dirección fija para obtener el valor. La utilidad de esta variable radica en que su valor esta en una dirección conocida y fija, la cual se hace útil al momento de necesitar algún modificador especial o si el valor necesitara ser cambiado en tiempo de ejecución, mediante un programa especial de la flash, lo cual repercute un cambio en la variable aunque esta se declare como constante, sin perder el valor cuando la energía del sistema falle o se genere un *reset*.

Ejemplos de declaración de este tipo de variables:

```
const unsigned char peso_maximo=123;  
  
const unsigned char temperatura_minima=12;
```

Declaración de constantes enumeradas enum

Las enumeraciones **enum**, posibilitan una forma rápida y conveniente de asociar valores constantes con un nombre de forma similar al #define, con la ventaja que las constantes bajo el **enum**, están en un mismo grupo y pueden tomar valores consecutivos o interrumpir la secuencia en alguna constante.

Su forma general de declaración es:

```
enum                                         tipoEnum{  
    NOMBRE_0 [= CONST_0], NOMBRE _1 [= CONST_1],  
    NOMBRE _2 [= CONST_2], NOMBRE_3 [= CONST_3]  
    ...  
    NOMBRE      _n      [=      CONST_n],  
};
```

Donde:

tipoEnum: es el nombre del tipo enumerado.

NOMBRE_i: es el nombre a declarar.

CONST_i: es la equivalencia de NOMBRE_i. Por aparecer dentro de los corchetes [], indica que son opcionales.

Si CONST_1 es omitida será por defecto 0 (cero) y la enumeración continuará de forma secuencial, es decir, que NOMBRE_1 será =1,

NOMBRE_2 será =2 y así sucesivamente. Pero si en algún momento el programador rompe la secuencia, ésta será inicializada en ese lugar.

La declaración anterior es equivalente a:

```
#define NOMBRE_0 CONST_0
```

```
#define NOMBRE_1 CONST_1
```

```
#define NOMBRE_2 CONST_2
```

```
...
```

```
#define NOMBRE_n CONST_n
```

Ejemplo práctico de declaración de enum es el siguiente:

```
enum meses{  
    ENE=1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC  
};
```

En este caso el nombre FEB tomará el valor de 2, MAR tomará el valor de 3 y así sucesivamente.

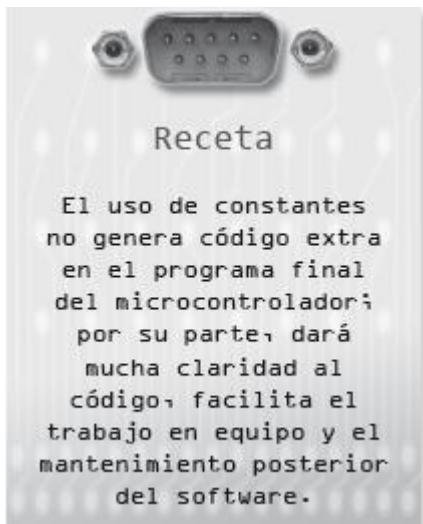
Constantes especiales tipo carácter

En el C se tienen definidos implícitamente todos los caracteres ASCII por su valor correspondiente decimal (o hexadecimal), esta definición facilita el manejo cuando de operar con cadenas se trata ver Anexo No. 1, también se tienen definidos los caracteres especiales tales como el <CR>, avance de línea <LF> y otros listados en la tabla 4.1 siguiente.



Constantes de caracteres especiales en C.

Código en C	Significado	Valor Hexadecimal	Valor Decimal
'\b'	Espacio Atrás <BS>	0x08	8
'\r'	Enter <CR>	0x0D	13
'\n'	Retorno carro <LF>	0x0A	10
'\0'	Nulo <NUL>	0x00	0
'\f'	Salto de Página <FF>	0x0C	12
'\t'	Tabulación Horizontal <TAB >	0x09	9
'\"''	Comilla doble	0x22	34
'\"'	Comilla sencilla	0x27	39
'\v'	Tabulación vertical <VT>	0x0B	11



Receta

El uso de constantes no genera código extra en el programa final del microcontrolador; por su parte, dará mucha claridad al código, facilita el trabajo en equipo y el mantenimiento posterior del software.

Para hacer uso de las constantes basta con encerrar entre comillas sencillas, el carácter ASCII en cuestión, de esta forma ‘a’ indicará el valor numérico del ASCII de la letra a siendo en este caso 0x61 en hexadecimal ó 97 en decimal; en otro caso ‘C’ sería 0x43 en hexadecimal ó 67 en decimal, evitando de esta forma tener que recurrir a la conversión ASCII o a la memoria del programador que no es de confiar en algunos casos.

Para referencia la tabla ASCII se encuentra también en el Anexo # 1.

EJEMPLO No. 4

Declaración y uso de constantes en C

Objetivo:

El objetivo es realizar un ejemplo que declara como macro el encendido del Led OUT-1 y una constante que apaga el Led OUT-1 del PIC-Link.

Declarar también en memoria flash una constante que contiene un valor, si el valor de la constante es mayor a 128, el Led OUT-1 encenderá, de lo contrario encenderá únicamente al inicio del programa y luego quedará apagado continuamente.

Ejecutar el programa con la opción paso por paso en ensamblador para visualizar los estados de la salida OUT-1.

Solución:

El proyecto solo contiene un archivo fuente main.c, y se ubica el cursor en la función main(), el cuerpo está listado a continuación:

```
***** Ejemplo 4 *****
// Declaración y Uso de constantes en C
// Fecha: Marzo 28 ,2009
// Asunto: Encendido LED OUT-1 usando CCS Para PIC,
//         Usando definición de constantes.
// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)
//           para Microcontrolador Microchip™ 16F877A.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#include "constantes.h" /*incluye archivos de configuración*/
#define OUT_1_On() output_high(PIN_B4)
#define OUT_1_Off() output_low(PIN_B4)
#define VALOR_MAX_ON 128 //valor de la constante
const unsigned char valor_en_flash=120;
void Mcu_Init_PIC(void){
    port_b_pullups(TRUE);
    setup_adc_ports(NO_ANALOGS);
    setup_adc(ADC_OFF);
    setup_psp(PSP_DISABLED);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC);
    setup_vref(FALSE);
}
void main(){ // Función Principal
Mcu_Init_PIC();
// TODO: USER CODE!!
OUT_1_On(); //enciende el led
if(valor_en_flash <= VALOR_MAX_ON){
    OUT_1_Off(); //apaga el led
}
for(;;){ } //loop infinito
}
Se finaliza la edición del programa.
```

Compilación:

Se compila con la presión del botón F9, el compilador no deberá mostrar errores de sintaxis, de no ser así, se debe verificar la línea de error y comparar el código digitado con el mostrado en el código fuente anterior.

Depuración del programa:

Realizar ahora la conexión del sistema PIC-Link a la computadora y realizarlo en modo

DEBUG  con el botón “Single Step” para observar claramente el comportamiento de cada línea.

Discusión:

Al declarar un valor constante como lo es **VALOR_MAX_ON** este valor es reemplazado en el código cada vez que es invocado, sin ninguna alteración, y tampoco generará código mayor si el valor es usado directamente en la línea así:

```
if(valor_en_flash <= 128){
```

Sin embargo, el código quedará más fácil de entender y fácil de modificar en un futuro con el uso de la constante declarada.

Nótese además, que a la variable “**valor_en_flash**” le asigna una dirección que dentro del mapa de memoria del microcontrolador es memoria FLASH, dado que su valor será constante, solo podrá ser leído, más no escrito.

De igual forma, una vez se invoca **OUT_1_On()** o **OUT_1_Off()**, esta secuencia es reemplazada por su equivalencia a la derecha del #define.

Tampoco deberán aparecer mensajes de precaución (warnings) debido a que todas las variables están siendo utilizadas de forma adecuada.

El único posible error que puede surgir al crear el proyecto es con la función
setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1); la cual es generada por el compilador CCS con doble paréntesis y punto y coma al final, lo que origina en el compilador el Error 51 : señalando el final de la línea, esto es solucionado solamente removiendo un paréntesis y un punto y coma.

4.3.2 Variables

Las variables son localizaciones de memoria que contienen algún valor. El programa podrá usar su valor o su dirección para acceder o modificar el dato que contiene. Dependiendo de los valores máximos y mínimos que ha de tomar una variable, se hace su declaración, y se define su signo dependiendo si tomará valores solo positivos o tomará tanto valores positivos como negativos.

Para declarar o separar espacio para una variable se debe elegir un nombre; acá es importante considerar un nombre que represente realmente lo que almacena, por sintaxis deberá empezar siempre por una letra y puede tener combinaciones alfanuméricas incluyendo el carácter ‘_’ (*underscore*), es útil por práctica de programación en equipo, que la primera letra sea minúscula, de esta forma permitirá al programador conocer en cualquier parte del programa donde este su nombre, que se trata de una variable (sin tener que regresar a su declaración para conocerlo).

Al momento de declarar una variable se deben considerar dos aspectos: El signo de la variable: es decir, ¿la variable tomará valores positivos, negativos o ambos?, si para el caso es indiferente se selecciona el positivo, los procesadores de 8 y 16 bits ejecutarán más rápido operaciones con valores positivos (incrementos, decrementos, comparaciones, etc.), que con los valores negativos.

Tamaño: ¿cuál es el valor mínimo y máximo que tomará la variable? Se deberá seleccionar el tipo mínimo que cubra todos los valores de la variable.

Dependiendo de la longitud en bytes requeridos para almacenar todos los posibles valores de la variable se puede hacer uso de los tipos de almacenamiento de datos según el ANSI C.

4.4 Tipos de datos

4.4.1 char en Codewarrior ® (o int8 en CCS)

(carácter), reserva 1 byte en memoria, la variable de este tipo puede tomar 256 posible valores, si es positiva desde 0 hasta 255, si es signada desde -128 hasta +127.

4.4.2 int en Codewarrior ® (o int16 para CCS)

(entero), reserva 2 bytes continuos en memoria para el almacenamiento de la variable. Este tipo puede tomar valores desde 0 hasta 65535 si la variable es positiva y desde -32.768 hasta +32.767 si la variable es signada.

4.4.3 long en Codewarrior ® (o int32 para CCS)

Reserva 4 bytes continuos en memoria para almacenar el valor de la variable. En valor con signo va desde -2.147.483.648 a +2.147.483.647, en valor no signado va de 0 a 4.294.967.296 (2^{32})

4.4.4 float (en Codewarrior ® y en CCS)



El tipo de datos flotante IEEE32, reserva 4 bytes en memoria.

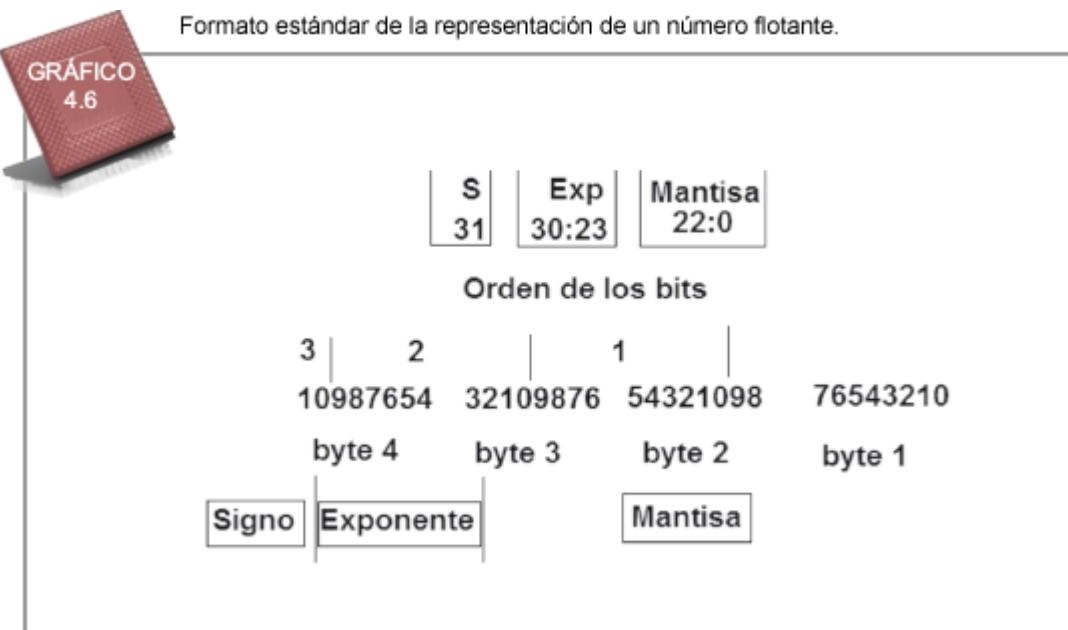
Las variables del tipo float se representan por medio de una mantisa, que es un número entre 0.1 y 1.0 que se multiplica por un exponente de una potencia de 10.



Estas variables se representan por medio de la mantisa, que es un número entre 0.1 y 1.0, y un exponente que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número deseado.

Tanto la mantisa como el exponente pueden ser positivos o negativos.

De los 32 bits utiliza 24 para la mantisa (1 bit para el signo y 23 para el valor) y 8 para el exponente (1 bit para el signo y 7 para el valor).



Para el tipo float IEEE32 el número mínimo es 1.17549435 E-38 y el máximo 3.402823466 E+38.

Así por ejemplo, el número **+1** se representa así:

En hexa: **0x7F 0x000000**

Signo: **0** (positivo).

Exponente: **0x7F** (Offset)

Mantisa: **0x000000**, la cual se compone de la siguiente manera: representación binaria del valor absoluto del número. Lo rota a la izquierda hasta encontrar un 1 → 1.00000 acá para la rotación y se completa con ceros a la derecha hasta completar los 23 bits.

4.4.5 double en Codewarrior ® (no soportado en CCS)

Representa una variable de punto flotante de doble precisión. Reserva 8 bytes (64 bits) continuos en memoria.

Se representan por una **mantisa** de 53 bits (1 para el signo y 52 para el valor).

El exponente 11 bits (1 para el signo y 10 para el valor).

Para el tipo double IEEE64 el número mínimo será 2.2259738585972014E-308, mientras el máximo será 1.7976931348623157E+308.

4.5 ¿CÓMO ELEGIR UN TIPO DE VARIABLE EN UN SISTEMA EMBEBIDO?



La clave para elegir el tipo de dato que se asignará a una variable consiste en determinar su valor mínimo y máximo, y proceder a elegir el dato más pequeño que pueda representar este rango de valores.



En los sistemas embebidos de 8 bits los datos usados como variables suelen ser de 8 bits, ya que su longitud coincide con la del bus de datos.

Al momento de elegir el tipo de dato que se asignará a una variable, deberá preguntarse por su valor **mínimo** y **máximo** posible, a fin de elegir el tipo más pequeño que puede representar todos sus posibles valores, esto permitirá que el almacenamiento y uso de la memoria sea el menor posible y además que el procesamiento de la misma tome el menor recurso del procesador durante su ejecución.

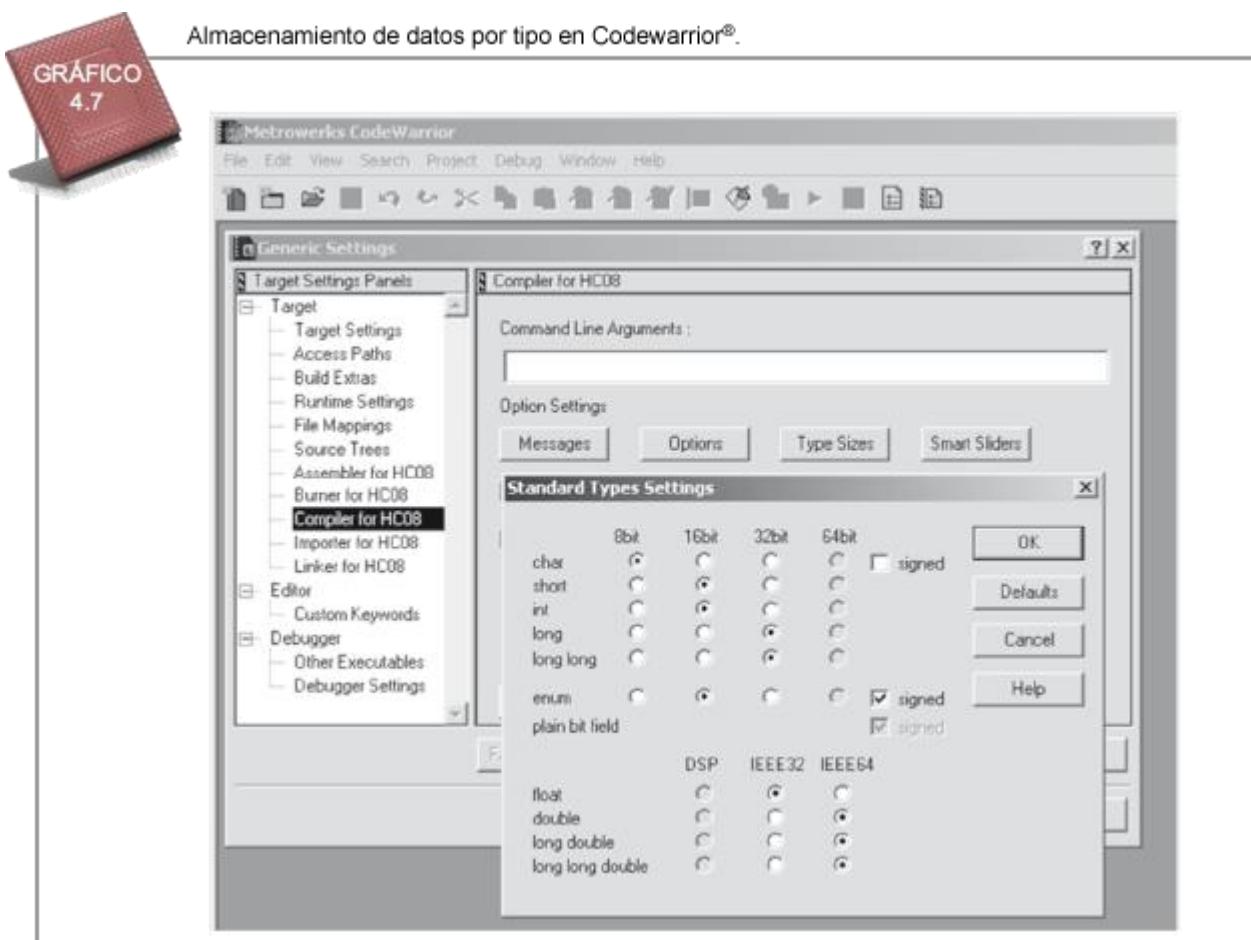
El tipo de datos preferido en los sistema embebidos de 8 bits es el tipo “char no signado”, este tipo será el que el procesador opere de manera más eficiente, dado que su longitud coincide con la longitud del bus de datos, la naturaleza del procesador, será entonces el tipo de datos preferido, aunque el preferido y por defecto del ANSI C es del tipo “int signado”.

La reserva de los bytes que ocupa un tipo de datos se hace de manera secuencial dentro del mapa de memoria, ocupando los bytes de mayor peso la dirección más baja en memoria.

De esta forma una variable de tipo **int** ocupara 2 bytes seguidos en memoria, donde al byte de mayor peso, le corresponde la menor dirección m, y la parte baja del **int** ocupará la dirección siguiente m+1.

El compilador de Codewarrior® tiene una ventana con los tipos estándar disponibles (*Gráfico 4.7*). Esta ventana solo se recomienda abrir en caso de necesitar consultar los tipos, mas no se recomienda modificar los valores por defecto (*Default*), y en lugar de esto se recomienda que cada variable este acompañada explícitamente por los modificadores que le corresponden.

¿La Razón? Uno de los programadores del equipo de trabajo podría declarar sus variables por defecto como **signed**, mientras que otro miembro del equipo podría modificar la opción para que por defecto sus variables sean **unsigned**, y con esto el código que tienen ambos no podría compartirse porque no conservarían el signo y dependerían de la opción seleccionada en esta ventana.



4.6 MODIFICADORES A TIPOS DE DATOS COMUNES EN SISTEMAS EMBEBIDOS

Los tipos de datos al momento de realizar su declaración, pueden ir acompañados de una o varias palabras reservadas del ANSI C, las cuales le proporcionan **atributos especiales** a la variable, sin cambiar la medida en bytes que ocupa, pero sí cambian el tratamiento y la forma como son manejadas y optimizadas.

Estos modificadores deben ir antes del tipo de dato. Para el compilador de C el modificador brinda información adicional sobre la variable, con el objetivo de darle un tratamiento adecuado cuando

genera el código en ensamblador que va a operar la variable, de esta manera produce código más óptimo, veloz y robusto.

La utilidad del modificador radica también en operaciones de variables que al momento de incurrir en una operación resulten ser de diferente naturaleza, de allí que sea necesario hacer un molde (*cast*) que permita a una variable declarada ajustarse a la naturaleza de otra variable.

El concepto, uso y razón de los modificadores se tendrán más claro con la definición de cada uno de ellos, los cuales se listan a continuación:

4.6.1 El modificador “**unsigned**”

Para generar un código más compacto y eficiente es recomendable utilizar el modificador “**unsigned**”, encargado de interpretar el bit más significativo del dato como valor y no como signo.

El modificador “**unsigned**” le proporciona a la variable la característica positiva o, lo que es lo mismo, que su menor valor será 0 (cero), mientras su valor máximo será el de todos sus bits en 1 (caso **unsigned char** su valor máximo será 255). El modificador **unsigned** interpreta el bit más significativo del dato como valor y no como signo.

Este modificador debe ser considerarlo por defecto, porque le permitirá al compilador generar un código más compacto y eficiente. Aunque ciertas arquitecturas como la del HC(S)08 de Freescale™ adicionó la bandera V (sobre flujo de complemento a dos), que permite el manejo signado de forma más rápida, el tipo de datos positivo sigue siendo el tipo de datos más eficiente para el manejo de una variable, de allí que si al momento de declarar la variable es indiferente el signo que esta manejará, se debe decidir que esta sea positiva, con lo cual su declaración y manejo será mejor y más sencillo para la máquina.

El tipo de datos preferido en un sistema embebido de 8 bits será el “**unsigned char**”, por ser positivo y ser de menor longitud en bytes.

4.6.2 El modificador “**signed**”



El modificador “**signed**” le indicará al compilador que el bit más significativo de la variable corresponde al signo (1: negativo, 0: positivo), los bits restantes corresponden al complemento a dos del valor absoluto del valor de la variable.

El ANSI C define este modificador por defecto, de tal manera que si una variable se declara sin el modificador de signo, asumirá el “**signed**”, algunos ambientes de compilación permiten cambiar esta opción por defecto (*Gráfico 4.2*), sin embargo, no es recomendable acudir a ella y en su lugar se recomienda al momento de la declaración, ser específicos en el signo sin asumir los valores por defecto, de esta forma se independiza de la configuración específica del ambiente de programación, creando así un código mas portable.

Resumen de tipos de datos en memoria:

RANGOS DE VALORES DE LOS TIPOS DE DATOS USANDO EL MODIFICADOR DE SIGNO

TABLA 4.2

Tipo Declarado	Nro. Bits	Rango
unsigned char	8	0 a + 255 (TIPO DE DATOS PREFERIDO EN 8 BITS)
signed char	8	-128 a +127
char	8	-128 a +127
unsigned int	16	0 a +65535
signed int	16	-32768 a +32767
int	16	-32768 a +32767
unsigned long	32	0 a +4294967295
unsigned long	32	- 2147483648 a + 2147483647
float	32	1.17549435 E-38 a 3.402823466 E+38
double	64	2.259738585972014E-308 a 1.7976931348623157E+308

4.6.3 El modificador “volatile”

El modificador “**volatile**” de variables, le indicará al compilador que el valor de una variable podría cambiar por medios no especificados explícitamente por la secuencia lógica del programa, en su lugar un agente externo, como puede ser el hardware externo o una interrupción, podría cambiar su valor.



Si una variable va a ser manipulada durante una interrupción, es adecuado que cada vez que el programa haga una referencia a ella, su valor no sea asumido por defecto sino que se accese a la dirección donde se encuentra dicha variable (*volatile*).



Por tal razón cada vez que el programa haga una referencia a su valor no deberá asumirlo, sino que deberá hacer acceso (sea de lectura o escritura) a la dirección de la variable.

El compilador no optimizará ningún uso de la variable y transportará línea por línea de C al ensamblador tal cual como fue digitado el código.

El uso de este modificador se hace útil para todas aquellas variables que van a ser **manipuladas dentro de una interrupción**, ya que esta puede ocurrir en lugares no predecibles del código, y por tal motivo no se puede asumir valores pasados de la variable.

También se obliga su uso a aquellas variables cuyo valor dependa del hardware externo conectado al microcontrolador (como son los puertos) o registros internos del mismo.

Para ilustrar el uso de este modificador y sus consecuencias en el código generado considérese el ejemplo No. 5:

EJEMPLO No. 5

Usos y efectos del modificador “volatile”

Objetivo:

Entender el efecto del modificador “volatile” en el código generado en ensamblador.
El programa realiza diferentes asignaciones a una variable en RAM llamada vble1.

Solución:

Edición del programa:

Parte A: Sin “volatile”

A continuación se presenta el código sin el uso del modificador “volatile” que habilita las optimizaciones obvias:

linea#	Código en C	Código Generado en Ensamblador
línea 1	unsigned char vble1;	vble1 equ \$80
línea 2	void main(void){	main:
línea 3	vble1 = 149;	→ MOV #149T,vble1 ;código no generado
línea 4	vble1 = 43;	→ MOV #43T,vble1 ;código no generado
línea 5	vble1 = 49;	→ MOV #49T,vble1

```

línea 6  if(vble1 > 10){ → LDA vble1      ;código no generado, ya que la
          condición SIEMPRE es Verdadera
          → CMPA #10T      ; código no generado
          → BMI label1     ;código no generado
línea 7  PORTA = vble1;   → MOV vble1,PORTA
línea 8  }else{           label1:
línea 9  PORTA = 0;      → CLR PORTA      ;código no generado
línea 10 }
línea 11 vble1 = vble1 + 1; → INC vble1
línea 12 vble1;          → LDA vble1      ; código no generado
línea 13 for(;;){        → label2: BRA label2: ;loop infinito
línea 14 }
línea 15 }               → RTS

```

Parte B: Con “volatile”

A continuación el código en C y su respectivo generado en ensamblador con el uso del modificador “volatile”, que no optimizará el uso de la variable vble1.

Línea #	Código en C	Código Generado en Ensamblador
línea 1	volatile unsigned char vble1;	vble1 equ \$80
línea 2	void main(void){	main:
línea 3	vble1 = 149;	→ MOV #149T,vble1
línea 4	vble1 = 43;	→ MOV #43T,vble1
línea 5	vble1 = 49;	→ MOV #49T,vble1
línea 6	if(vble1 > 10){	→ LDA vble1 → CMPA #10T → BMI label1
línea 7	PORTA = vble1;	→ MOV vble1,PORTA
línea 8	}else{	label1:
línea 9	PORTA = 0;	→ CLR PORTA
línea 10	}	
línea 11	vble1 = vble1 + 1;	→ INC vble1
línea 12	vble1;	→ LDA vble1
línea 13	for(;;){	label2: BRA label2:
línea 14	}	
línea 15	}	→ RTS

Discusión:

Si se analiza la Parte A, en la cual NO se usa el modificador “volatile”, se hace la declaración de una variable llamada vble1, el compilador no generará código para las líneas 3 y 4 debido a que no le ve sentido en asignarle los valores 149 y 43 dado que en la línea 5 se le asignará finalmente el valor 49, por esta razón solo generará el código para la línea 5.

En la línea 6 encuentra un condicional “if” el cual, en tiempo de compilación, deduce que la condición (vble1 > 10) es SIEMPRE verdadera, porque vble1 es 49, por esta razón puede eliminar la evaluación en tiempo de ejecución y eliminar el código equivalente al “else”, porque la condición nunca será falsa, y por esta razón optimizará y eliminará el código equivalente de la línea 9.

Seguido el código de la línea 11 realiza el incremento de la variable, pero en la línea 12 elimina el código generado debido a que no encuentra utilidad en hacer un acceso de lectura a la variable vble1.

Pero:

Si se considera que la variable vble1 se manipulará dentro de una interrupción, la cual inicializa la variable en 0 (cero), así:

```
interrupt void ISR(void){  
    vble1 = 0;  
}
```

Se asume ahora que la interrupción ocurre cuando el código finaliza la línea 5 (en la cual la vble1 contiene el valor 49) y con esto regresará de la interrupción a la línea 6.

De ser así y aunque vble1 se modificó al valor 49 en la línea 5, al regresar a la línea 6 el valor es de 0 (cero) y con esto la condición y evaluación del “if” sería falsa y la optimización del compilador NO generaría código para esta línea por considerarlo redundante.

Ahora, el código en la Parte B es idéntico al anterior solo que a la variable se le antepone el modificador “volatile”.

Como se puede observar, el código generado ocupará más espacio en memoria debido a que no se optimiza

ningún acceso de lectura ni de escritura de la variable volatile vble1. Con esto se ejecuta y se evalúa en todo momento que requiere hacer evaluación o asignación de la variable.

El código será más extenso, pero será completamente seguro y conserva la lógica que el programador realizó al momento de digitar el código.

Todas las variables que van a ser accesadas en interrupción deberán estar acompañadas del modificador “volatile”, esto previene que el compilador asuma y optimice código en tiempo de ejecución que considere redundante. También deben tener el modificador “volatile” todos los registros internos y puertos del microcontrolador, dado que su valor puede cambiar por acciones externas al código en sí.

Es posible usar los modificadores “const” y “volatile” sobre una misma variable, la primera le indicará al compilador que la variable es constante y que estará ubicada en memoria flash, pero al mismo tiempo le indica que **NO** debe optimizar su manejo; además, si una asignación se hace sobre la variable, el compilador generará un ciclo de escritura. Este tipo de modificadores juntos se usa en variables que puedan modificarse con un ciclo de lectura, como el registro SCS1 (*Serial Control and Status 1*).

4.6.4 El modificador “near”

El modificador “near” le solicita al compilador asignarle a la variable una dirección dentro del rango de direcciones directa del mapa de memoria del microcontrolador, o lo que es lo mismo, una dirección cuya parte alta o de mayor peso sea 0 (cero) (direcciones menores a 0x0100 que correspondan a la RAM) (*ver Gráfico 2.14 y gráfico 4.8*).

La utilidad de este tipo de variable radica en que su manipulación será muy eficiente, porque para realizar un acceso a la variable, éste se realizará mediante direccionamientos directos, los cuales son muy eficientes tanto en tiempo de ejecución como en el espacio ocupado en memoria de programa.

Es recomendable el uso del modificador “near” sobre aquellas variables que serán de uso muy frecuente en el programa porque permiten optimizar el código, también para aquellas variables que requieren acceso rápido en zonas críticas del programa o rutinas que se necesitan ejecutar de manera rápida.

En el ejemplo No. 6 se ilustra el uso de este modificador.

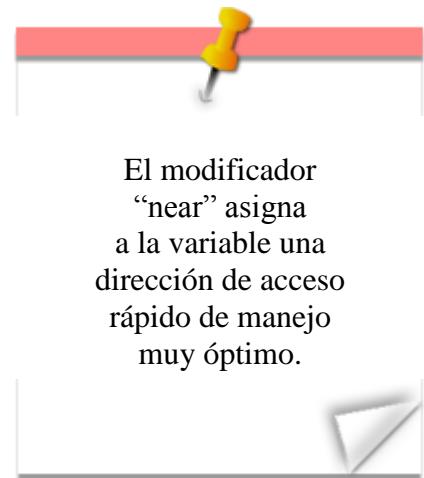
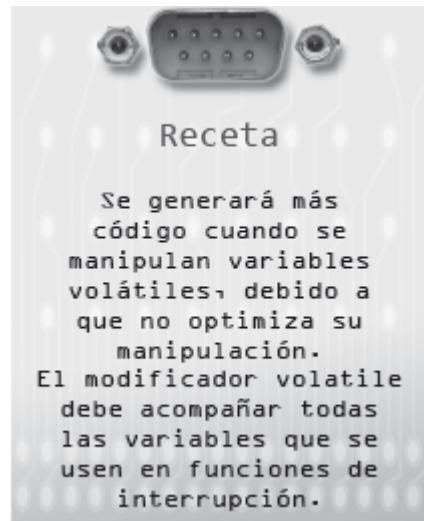
EJEMPLO No. 6

Usos y efectos del modificador “near”

Objetivo:

Entender el efecto del modificador “near” en el código generado en ensamblador.

El programa realiza diferentes asignaciones a variables en RAM llamadas vble1 y vble2.



Solución:

línea # C ódigo en C Código Generado en Ensamblador Comentario

línea 1 **near volatile unsigned char vble1;** vble1 equ \$80 ;Asignación RAM en la zona directa

near volatile insigned char vble2; vble2 equ \$81 ;Asignación RAM en la zona directa

línea 2 **void main(void){** main:
línea 3 **vble1 = 0x5F;** mov #\$5F,vble1 ;3 bytes @ 4 ciclos de ejecución
línea 4 **vble2 = 0;** clr vble2 ;2 bytes @ 3 ciclos de ejecución
línea 5 **vble1++;** inc vble1 ;2 bytes @ 4 ciclos de ejecución
línea 6 **vble2 = vble1;** mov vble1,vble2 ;3 bytes @ 5 ciclos de ejecución
línea 7 **for(;;){}** loop: bra loop
línea 8 **}**

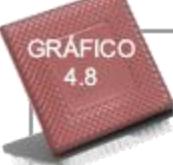
Total 10 bytes @ 16 ciclos de ejecución.

Discusión:

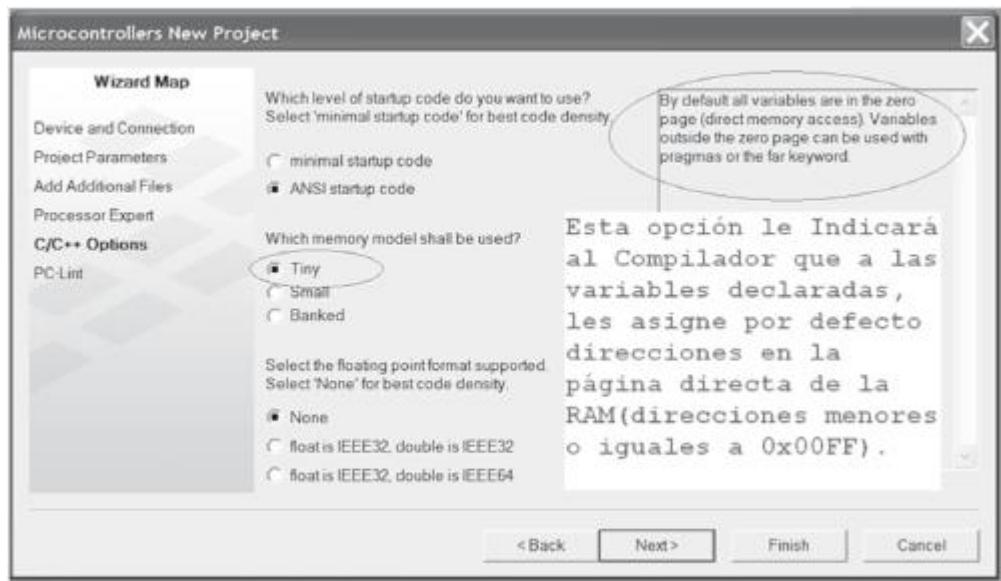
A todas las variables declaradas acompañadas por el modificador “**near**”, se les asigna una dirección que está en el rango de RAM de la zona directa: menor a 0x00FF.

Para este caso la variable vble1 recibe la dirección 0x80 y vble2 recibe la dirección 0x81, facilitando de esta forma el uso de direccionamientos directo, como se hace en la línea 4 y línea 5, y el uso de mov en las líneas 3 y 6.

El total en bytes en memoria de programa es de 10 bytes, que suman 16 ciclos en tiempo de ejecución.



Opción del proyecto para asignar variables NEAR en RAM por defecto.



Esta opción le Indicará al Compilador que a las variables declaradas, les asigne por defecto direcciones en la página directa de la RAM(direcciones menores o iguales a 0x00FF).

4.6.5 El modificador “far”

El modificador “**far**” asignará a la variable definida una dirección fuera de la zona directa de memoria o zona extendida (direcciones mayores a 0x00FF). En este caso el compilador deberá accesar la variable con direccionamientos extendidos, los cuales típicamente ocupan 3 bytes de longitud (1 byte opcode + 1 byte dirección alta + 1 byte dirección baja) (*ver Gráfico 4.9*).

En el ejemplo No. 7 se ilustra su efecto en el código generado.

EJEMPLO No. 7

Usos y efectos del modificador “far”

Objetivo:

Entender el efecto del modificador “far” en el código generado en ensamblador.

El programa realiza diferentes asignaciones a variables en RAM llamadas vble1 y vble2.

Solución:

linea # Código en C	Código Generado en Ensamblador	Comentario
---------------------	--------------------------------	------------

línea 1 **far volatile unsigned char vble1;** vble1 equ \$100 ;Asignación de RAM la zona extendida **far volatile unsigned char vble2;** vble2 equ \$101 ;Asignación de memoria en la zona extendida

línea 2 **void main(void){** main:

línea 3 **vble1 = 0x5F;** lda #\$5F ;2 bytes @ 2 ciclos de ejecución

```

        sta vble1      ;2 bytes @ 4 ciclos de ejecución
línea 4  vble2 = 0;    clra      ;1 byte @ 1 ciclo de ejecución
            sta vble2      ;2 bytes @ 4 ciclos de ejecución
línea 5  vble1++;    lda vble1   ;2 bytes @ 4 ciclos de ejecución
            inca       ;1 byte @ 1 ciclo de ejecución
            sta vble1   ;2 bytes @ 4 ciclos de ejecución
línea 6  vble2 = vble1;  lda vble1   ;2 bytes @ 4 ciclos de ejecución
            sta vble2   ;2 bytes @ 4 ciclos de ejecución
línea 7  for(;;){}     loop: bra loop
línea 8 }

```

Total 17 bytes @ 28 ciclos de ejecución.

Discusión # 7:

Se concluye que ante el mismo código mostrado en el Ejemplo 6 se genera mas código con el modificador **far (17 bytes con far Vs. 10 bytes con near)**, debido a que con variables extendidas no es posible usar los modos de direccionamiento directos, ni MOV. De la misma forma, el tiempo de ejecución es mayor con el uso de far que **near (28 ciclos con far Vs. 16 ciclos con near)**.

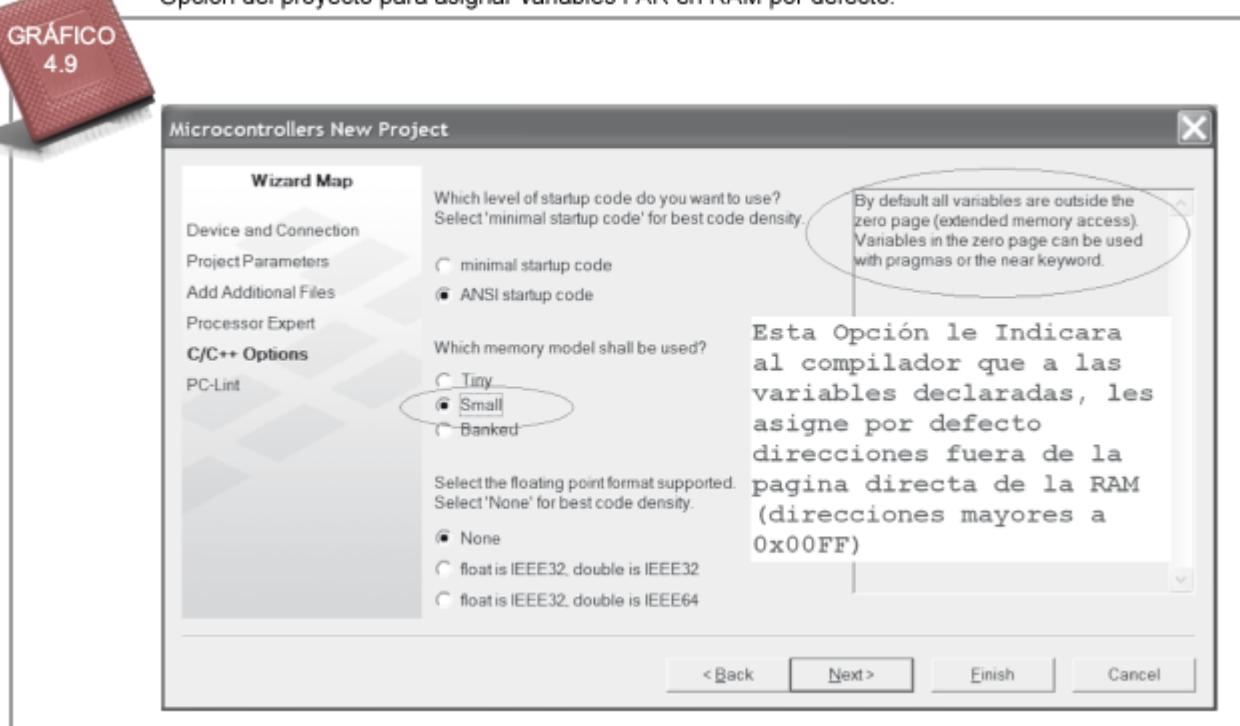


Es de esperarse que se debe posicionar el mayor número de variables posible en la zona **near**, a fin de generar código más pequeño en memoria y más rápido, pero como no es posible que TODAS las variables tengan cabida en esta zona, se recomienda usar por defecto la ventana del *Gráfico 4.9*, el cual declarará por defecto las variables por fuera de la zona **near**, y solo usar el modificador **near** para aquellas variables que son de uso frecuente o que su manejo requiere un código rápido.

La opción “near” debe reservarse exclusivamente para las variables más importantes o de mayor uso en el sistema. Para las demás se aconseja usar el modificador “far”.



GRÁFICO
4.9



4.6.6 El modificador “register”

El modificador “register” dedica uno de los registros del modelo de programación para la variable declarada. Se hace con el objetivo de realizar un acceso muy rápido a la variable en una función, debido a que resulta mucho más eficiente su acceso a que si estuviera declarada en la memoria RAM.

Este modificador solo aplica para las variables locales, nunca para las globales debido a que el compilador no puede prescindir de uno de los registros de trabajo exclusivamente a una variable.

El modificador es usado cuando se requiere realizar el manejo más óptimo posible, tanto en velocidad de ejecución como en espacio en memoria de programa sobre una variable local.

Para ilustrar este tipo de modificador y su efecto en el código generado, considérese el ejemplo No. 8 sobre una función de retardo.

EJEMPLO No. 8

Usos y efectos del modificador “register”

Objetivo:

Diseñar código de encendido y apagado periódico del led OUT-1 del AP-Link.

Realizar una función de retardo llamada **Delay**, la cual utiliza uno de los registros de la CPU para hacer su labor de conteo.

Usar el modificador “register” para este fin, y NO usar lenguaje ensamblador para su

implementación.

Solución:

Como primera medida se deben deshabilitar para éste y para los ejemplos posteriores el uso del COP ó *Watchdog*, a fin de no requerir resetearlo frecuentemente.

Se hace con un “1” en el bit COPD (COP Disable) del registro CONFIG1 del AP16A, de la siguiente forma:

```
#define Disable_COP() CONFIG1_COPD = 1 // deshabilita el COP
```

```
***** Ejemplo 8 *****  
//  
// Uso y Efectos del modificador “register”  
//  
// Fecha: Feb 4,2009  
// Asunto: Encendido y apagado de LED OUT-1  
// usando Codewarrior ®.  
//  
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)  
// para Microcontrolador Freescale™ AP16.  
// Version: 1.0 Por: Gustavo A. Galeano A.  
//  
*****
```

```
#include <hidef.h> /* for EnableInterrupts macro */  
#include "derivative.h" /* include peripheral declarations */  
  
//definiciones para encendido y apagado de OUT-1  
  
#define OUT_1_On() PTC_PTC3 = 1;DDRC_DDRC3 = 1  
#define OUT_1_Off() PTC_PTC3 = 0;DDRC_DDRC3 = 1  
  
#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP  
  
// Rutina Delay  
void Delay(void){  
register unsigned char vble_local; //declaración de variable  
unsigned int i;  
  
vble_local = 0; //inicializa la variable  
i = 0;  
while(i < 10000) i = i+1; //retardo  
while(vble_local < 200){ //mientras sea menor a 200  
vble_local = vble_local + 1; //la incrementa
```

```

        } //regresa al loop while
    }

void main(void){
    Disable_COP();
    EnableInterrupts;

    for(;;){
        OUT_1_On();
        Delay();
        OUT_1_Off();
        Delay();
    }
}

```

El código generado para la función Delay con la declaración anterior será:

Delay:

```

AIS #-2 ;separa espacio para la variable i
TSX
CLR 1,X ;borra parte alta de i
CLR ,X ;borra parte baja de i
BRA LE
L8: TSX
    INC 1,X ;incrementa variable i
    BNE LE
    INC ,X
LE: LDA ,X
    PSHA
    LDX 1,X
    PULH
    CPHX #10000 ;compara i con 10000
    BCS L8 ; i<10000?
    CLRA ;AccA usado para vble_local
L1: INCA ;incrementa AccA
    CMP #200 ;compara
    BCS L1 ;sino salta a L1
    A IS #2 ;recupera SP
    RTS ;retorna de la función

```

Para observar el funcionamiento puede compilar el programa, programarlo en el AP-Link y ejecutarlo.

Discusión:

Al usar el modificador “**register**” sobre la variable vble_local, el AccA (Acumulador A) será el recurso usado para contener a vble_local.

Acá se puede observar que con la instrucción CLRA se inicializa la vble_local en cero y la instrucción CMP #200 realiza la comparación de forma rápida, de forma diferente a como se hace el tratamiento de la variable local i, la cual está declarada de forma estándar (sin “register”), el código es mucho más rápido porque permite el uso de instrucciones inherentes.

4.6.7 El modificador “const”

El modificador “**const**” indica al compilador que la variable declarada no puede ser modificada o realizar cualquier acceso de escritura en la dirección de la misma.

El valor de la variable constante se define al momento de la declaración realizando su asignación.

Ejemplos de declaración con modificador const:

```
const unsigned char temperaturaMaxima=214;
```

Este tipo de declaración asigna a la variable temperaturaMaxima una dirección en memoria flash del microcontrolador y lo inicializa en el valor 214; sin embargo, ese valor pudiera ser alterado por un procedimiento especial de programación de la memoria de programa, el cual es un algoritmo especial para cada familia de microcontrolador.

Para una ilustración sobre el uso de este modificador puede referirse al Ejemplo # 4 en la página 167.

4.6.8 El modificador “static”

El modificador “**static**” aplicado a una variable la localiza en la **RAM fija** del mapa de memoria.



Cuando se requiere que una variable tenga un valor fijo, la opción “**const**” le indica al compilador que esa variable no podrá ser modificada de ninguna manera.



Para evitar que los procedimientos externos afecten la variable localizada en la RAM fija del mapa de memoria, se utiliza el modificador “**static**”.



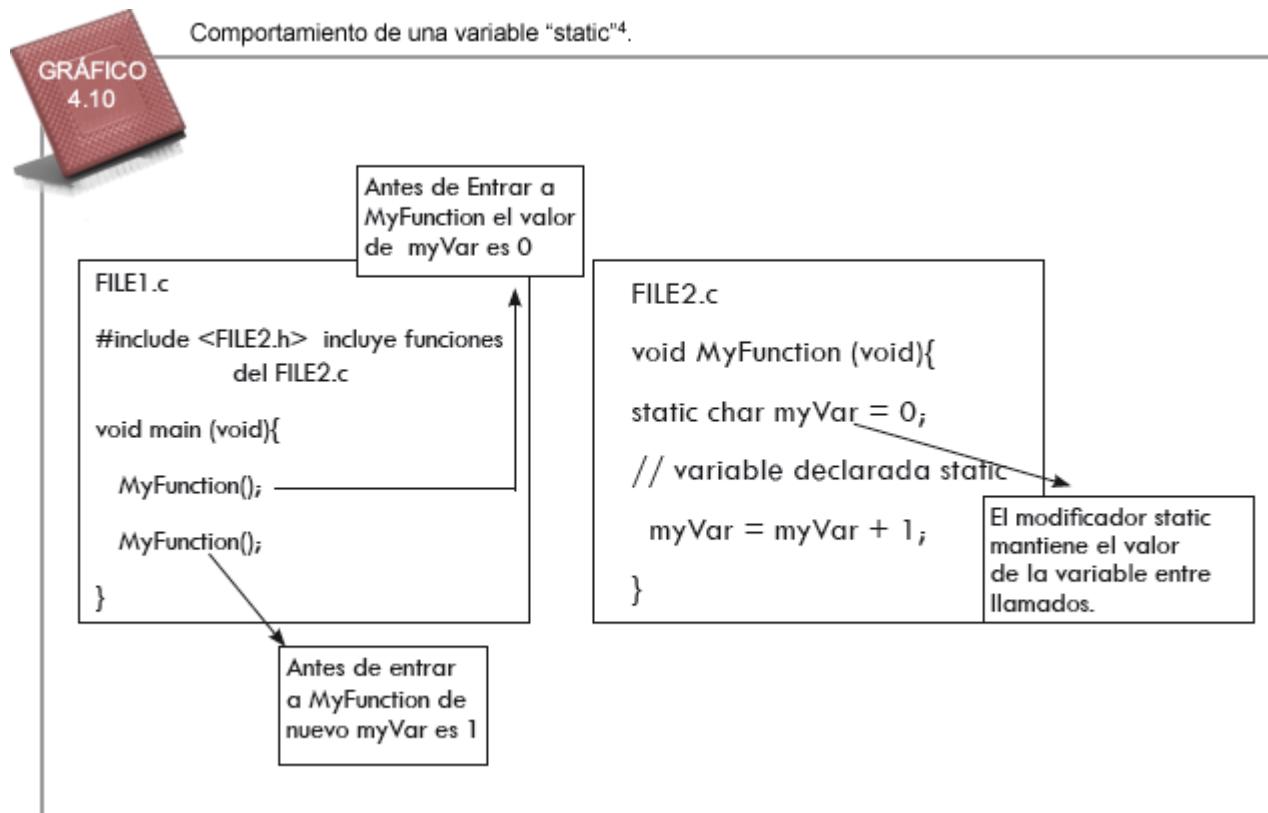
Con **static** una variable ocupa una **dirección única**, lo que significa que las localizaciones en memoria que ocupen, serán destinadas solo y únicamente a la variable. Además conservará su valor a menos que explícitamente se actúe sobre la variable, ningún otro agente o módulo podrá interactuar sobre ella.

El uso del modificador “**static**” además es útil para manejar el nivel de encapsulado y de visibilidad que tiene una variable o inclusive una función, con el objetivo de evitar que módulos externos a la variable pueda usarla (leerla o escribirla) y efectuar cambios que puedan alterar el funcionamiento de un módulo particular.

Toda variable que contenga este modificador no será reconocida fuera del módulo donde esta declarada, solo será reconocida por el módulo local.

Si la variable está declarada dentro de una función solo será reconocida y por lo tanto solo podrá ser operada dentro de la misma función, haciéndose invisible para el resto del módulo y por consiguiente para el resto del proyecto.

Si se consideran dos archivos del proyecto: FILE1.c y FILE2.c, ver *Gráfico 4.10* donde en FILE2.c → **MyFunction()**, la variable **myVar** es inicializada en 0 (cero), como cualquier variable global declarada y no inicializada. Al ser invocada **MyFunction()** por primera vez, el valor de **myVar** es incrementado en 1, al retornar de la función y entrar nuevamente, el valor de **myVar** conserva su valor anterior, debido a que NO está declarada en *stack* (variable local) sino que por el contrario, existe una dirección en la RAM fija que contiene y conserva el valor.



Objetivo:

Escribir un programa en C que invoque una función llamada Blink_OUT_1(), la cual a nivel interno realiza un ciclo de encendido y apagado de la salida OUT_1, solo las primeras 10 veces que la función es invocada.

Usar una función de retardo Delay() entre encendido y apagado de la salida, que permita visualizar el encendido y apagado una vez que el procesador está ejecutando el programa a máxima velocidad en modo RUN.

Usar variables internas a la función que no puedan ser accesadas desde el código principal del programa.

Solución:

El código completo se verá de la siguiente forma:

```
***** Ejemplo 9 *****
// Uso del modificador “static”
// Fecha: Marzo 28,2009
// Asunto: Encendido y apagado de LED OUT-1
// 10 veces usando CCS para PIC.
// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)
// para Microcontrolador Microchip™ 16F877A.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#include “static.h”
#define OUT_1_On() output_high(PIN_B4)
#define OUT_1_Off() output_low(PIN_B4)
#define NRO_MAX_BLINKS 10
void Delay(void){ //Función de retardo
unsigned long i;
i = 0;
while(i<65500){ i = i + 1; }
}
void Blink_OUT_1(void){ //Función apagado y encendido
static unsigned char nroBlinks; //Declaración de la variable
if(nroBlinks < NRO_MAX_BLINKS){
    OUT_1_On();
    Delay();
    OUT_1_Off();
    Delay();
    nroBlinks = nroBlinks + 1;
}
}
```

```
void Mcu_Init_PIC(); /*el cuerpo de la función en el Ejemplo 4*/
void main(){ // Función principal
    Mcu_Init_PIC();
    for(;;) { Blink_OUT_1(); } /* loop forever */
}
```

Discusión:

El modificador “**static**” sobre la variable **nroBlink** se encuentra dentro de la función **Blink_OUT_1()**, de modo que solo en esta función podrá ser leída o modificada.

La variable, a pesar de ser global y estar en zona de RAM fija, no puede ser accesada desde otra función.

Puede intentar agregar la línea:

`nroBlinks = 0;`
en el `main()` e intentar compilar, ante esto el compilador dará un error de variable no declarada:



La variable quedará protegida contra un acceso externo dada la declaración.

En caso de requerir usarla, la variable deberá declararse por fuera de la función, permitiendo así la visualización de la variable a la función `main()`.

Debe recordarse que las variables declaradas con este tipo de modificador ocupan un espacio fijo y por lo tanto el programa ocupara más espacio de la RAM si en su lugar se usan variables de tipo local.

Se puede observar que la variable tiene la visibilidad similar al de una variable local en el sentido que no puede ser accesada por una función externa, pero a diferencia de las variables locales, la variable estática mantiene su valor entre llamados a la función que la maneja, en este caso la función `Blink_OUT_1()` incrementa la variable en 1, y en el siguiente llamada a la función se conserva el último valor actualizado.

Las variables de tipo static son inicializadas en la rutina de inicio o de StartUp de forma similar a las variables globales y en caso de tener un inicializador este se realizará una sola vez en la rutina de inicialización y no cada vez que entra a la función que la usa, en este caso particular la variable es inicializada en cero antes de ir a la función `main()`.

En lo posible el programador solo debe usar el modificador para las variables que ameriten ser de este tipo, ya que en la mayoría de los casos es preferible usar variables locales que se crean y se destruyen en el stack, sin embargo en muchos casos por cuestión de velocidad de ejecución se puede recomendar el uso de una variable de tipo static, esto debido a que la manipulación de las variables fijas en RAM es mucho mas eficiente que el manejo de las variables locales creadas en el stack.

En muchas ocasiones y también con el ánimo de tener mayor eficiencia, pueden ir acompañadas del modificador near.

El modificador “static” no solo puede ser usado para las variables, también pueden acompañar a las funciones, con el ánimo de prevenir que módulos externos realicen accesos a estas funciones.

Cuando una función tiene el modificador “static” se denomina una función privada, a diferencia de cuando no lo tiene, se denomina una función pública.

4 Gráfico cortesía de Freescale™.

4.6.9 El modificador “extern”

Este modificador sobre una variable realiza una seudo-declaración que especifica una variable que está declarada en otro módulo o archivo del proyecto.

Se usa para indicar en un archivo que la declaración de la variable ya existe y está en otro módulo o archivo del proyecto.

Si se considera un proyecto que tiene varios módulos .C y .H, cada uno de los módulos contiene variables propias, las cuales se declaran mediante el modificador “static”. Sin embargo, existirán algunas variables que son globales y pueden modificarse o leerse en varios módulos, solo en uno de ellos se realizará la declaración de la variable, los demás, si desean usar esta variable, deberán tener la misma declaración antecedida del modificador “extern”.

Para ilustrar el uso del modificador considérese el ejemplo No. 10.

EJEMPLO No. 10

Uso del modificador “extern”

Objetivo:

Escribir un programa en C que realice lectura de la tecla INPUT-1, encienda el led OUT-2 al estar activa o presionada INPUT-1 y apague OUT-2 una vez INPUT-1 está inactiva.

Si la tecla es presionada más de 20 veces, la salida OUT-1 debe activarse.

Usar el modificador extern para la variable que realiza el conteo del número de presiones de la tecla INPUT-1.

Solución:

```
// Archivo teclado.h
#define OUT_2_On() output_high(PIN_B5) /*macro para encendido de Led2*/
#define OUT_2_Off() output_low(PIN_B5) /*macro para apagado de Led2*/
#define Key_Press() !input(PIN_B0)
extern unsigned char nro_presiones=0; /*declaración ficticia*/
void Verif_Teclado(void){ /*Función de verificar teclado,y manejo de estado*/
static char tecla_st=0;
    if(tecla_st){
        if(!Key_Press()){
            tecla_st = 0;
            OUT_2_Off();
            nro_presiones = nro_presiones + 1;
        }
    }else{
        if(Key_Press()){
            OUT_2_On();
            tecla_st = 1;
        }
    }
}

// Archivo principal del proyecto
//***** Ejemplo 10 *****
// Uso del modificador “extern”
// Fecha: Marzo 28,2009
// Asunto: Lectura de INPUT-1
// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)
// para Microcontrolador Microchip™ 16F877A.
// Version: 1.0 Por: Gustavo A. Galeano A.
//*****
#include "extern.h"
#include "teclado.c"
#define NRO_PRESS_MAX 20
#define OUT_1_Off() output_low(PIN_B4)
#define OUT_1_On() output_high(PIN_B4)
unsigned char nro_presiones=0; /*declaración real de la variable*/

void Mcu_Init_PIC(void); //Función completa en el Ejemplo 4
```

```

void main(){ // Función principal
    Mcu_Init_PIC();
    OUT_1_Off();
    for(;;) {
        Verif_Teclado();
        delay_ms (150);
        if(nro_presiones > NRO_PRESS_MAX){
            OUT_1_On();
        }
    } /* loop forever */
}

```

Discusión:

Para realizar el conteo del número de presiones de la tecla **INPUT-1** se cuenta con la función **Verif_Teclado()**, la cual es invocada periódicamente por la función main(), que se encuentra en el archivo **extern.c**.

Desde el archivo **extern.c** se determina si el número de presiones de la tecla **INPUT-1** es mayor a 20, para lo cual se debe accesar **la misma** variable modificada en el archivo **teclado.c**.

Bajo estas condiciones se requiere leer su valor, por lo tanto se necesita declarar la variable **nro_presiones** nuevamente en el módulo **extern.c**, pero se hace referencia a la misma variable que se está declarando en el módulo **teclado.c**; por esta razón en uno de los dos archivos, se usa el modificador **extern**, que indica que la variable va a tener una seudo-declaración. En este caso se realiza la declaración de la variable en el modulo **extern.c**, y se realiza la pseudo-declaración en el módulo **teclado.c**.

En todos aquellos módulos donde se realice la seudo-declaración mediante el modificador **extern**, se deberá tener el mismo tipo y modificadores que tiene la variable en el módulo donde se realiza la declaración completa.

Tal como se uso en este caso la variable que va a tener seudo-declaraciones deberá ser global y en ningún caso ser usado con el modificador static, pero debe usarse de forma responsable por todos los módulos que realicen el acceso a la variable debido a que la variable si bien es posible leerla, también es posible escribirla.

No es posible darle un acceso parcial o de solo lectura a la variable.

4.7 MOLDES O “CASTING” PARA VARIABLES Y CONSTANTES

Se ha visto que se tienen diferentes tipos de variables dependiendo de su longitud en bytes, de su localización en memoria y de sus modificadores, cada uno brindando una característica especial a cada una de ellas.

Considérese la igualdad $a = b$, que indica con claridad que el contenido de la variable “ b ” lo almacene además en la variable “ a ”. Esta igualdad es 100% efectiva y posible si la variable “ a ” es del mismo tipo y por tal motivo tiene la misma declaración que la variable “ b ”, de lo contrario, el compilador indicará un “warning” y en algunos casos un “error” que debe ser corregido para que la compilación sea realizada.

Para el caso del “warning” se da porque los tipos de variables son diferentes, sea por longitud en bytes de las variables o porque sus modificadores son distintos. En el caso que la variable “ a ” sea de una longitud menor que la variable “ b ”, el contenido de la variable “ b ” solo podrá ser almacenado de forma parcial, en este caso la parte alta será truncada y solo se almacenará la parte baja de “ b ”.



Para optimizar el código el compilador hace que las variables tenga la misma longitud, es como si se depositarán en un molde de igual tamaño; en el caso de variables distintas, el sistema completa con ceros la de menor longitud.



El programador puede determinar exactamente cómo será completada cada variable a fin de que se ajuste a lo esperado.



En el caso que la variable “ a ” fuera de longitud mayor que “ b ”, la parte alta será complementada con ceros. Para el caso de diferentes signos el ajuste que realice el compilador puede no ser conveniente o el esperado por el programador, por esta razón, en la mayoría de los casos cuando se realicen igualdades, es necesario ser explícitos en la conversión que se desea hacer, realizando el ajuste a la variable que se espera reciba el dato original. Este proceso se lleva a cabo anteponiendo el nuevo tipo de dato al que se quiera convertir el dato original mediante la siguiente sintaxis:

$a = (<\text{nuevo tipo}>)b;$

Para tener mejor claridad sobre el uso de los moldes y su operación examínense las siguientes situaciones, descritas de forma gráfica en los *Gráficos 4.11 y 4.12*.

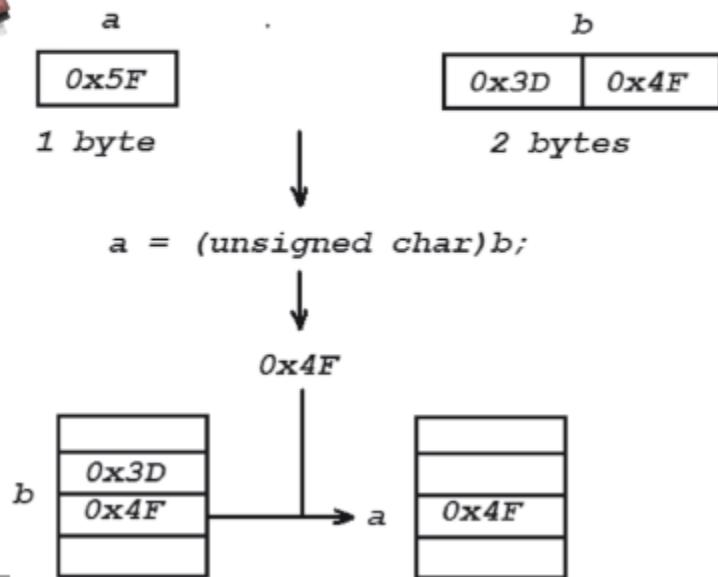
```
unsigned char a;
```

```
unsigned int b;
```

```
a = (unsigned char)b;
```



Molde o “casting” entre variables int a char.



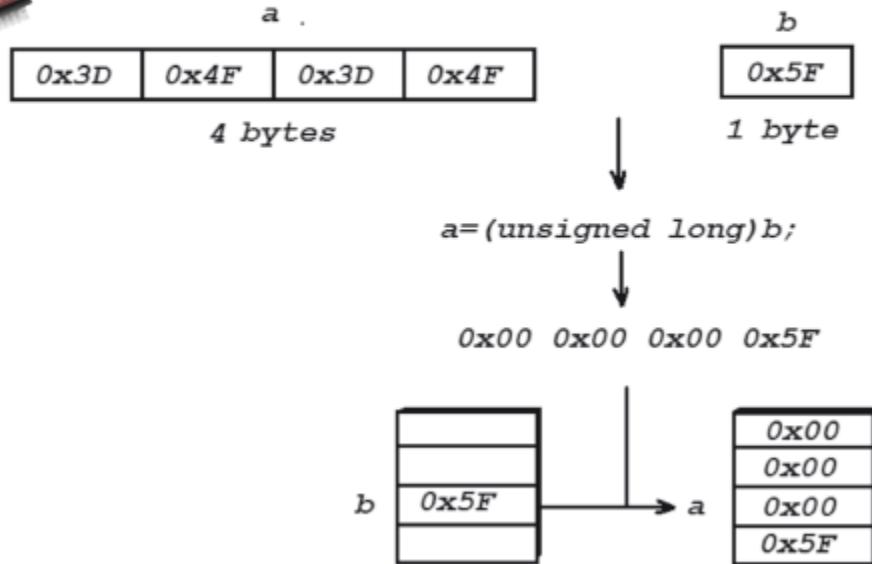
```
unsigned long a;
```

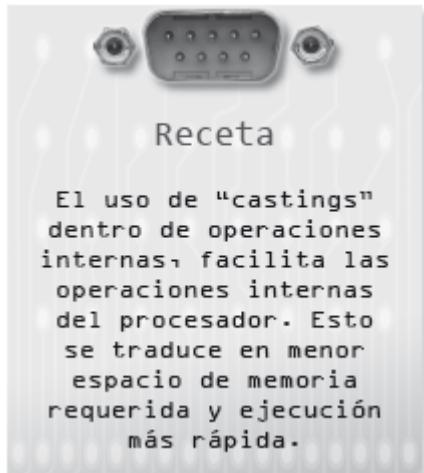
```
unsigned char b;
```

```
a = (unsigned long)b;
```



Molde o “casting” entre variables char a long.





Además, el uso de los moldes es útil entre expresiones para reducir la carga en operaciones matemáticas y lógicas, permitiendo que estas se realicen de forma rápida y con el menor procesamiento posible. El ajuste o moldeado se puede usar no solo para variables, sino para constantes.

El uso de "castings" dentro de operaciones internas, facilita las operaciones internas del procesador. Esto se traduce en menor espacio de memoria requerida y ejecución más rápida.

RESUMEN DEL CAPÍTULO

Después de la discusión sobre las ventajas y desventajas del lenguaje ANSI C sobre el lenguaje ensamblador, y antes de poder justificar el estudio detallado de los temas propios de C, se debe estar convencido que son superiores las ventajas que las desventajas del C, que la herramienta puede ayudar a que los proyectos futuros se puedan abordar por este camino y que en la medida que la programación sea muy prudente y conscientes de qué máquina se está usando, se puede hacer que las desventajas tengan un impacto menor sobre el desempeño, el consumo de energía y la velocidad de ejecución.

Para tranquilidad de los seguidores del C, el espacio en memoria cada vez incide menos en el costo final del microcontrolador; por poner un ejemplo, hasta hace solo tres años un microcontrolador de 32KBytes de memoria Flash costaba alrededor de USD 5.00, hoy en día por un procesador similar en memoria y recursos se paga alrededor de USD 3.00 y la tendencia es a bajar aún más.

La elección adecuada de la dimensión de una variable, es clave para la optimización de la memoria tanto de programa como RAM en el microcontrolador; el manejo, optimización y procesamiento de la variable es controlado por los modificadores que anteceden la variable al momento de su definición.

La elección correcta de una variable incorpora dos aspectos: el signo de la variable y el rango mínimo y máximo que tendrá, de esta forma se selecciona la más adecuada, siendo la más óptima para sistemas de 8 bits la variable unsigned char.

PREGUNTAS DEL CAPÍTULO

¿Cuál es considerada la mayor ventaja de la programación estructurada C sobre la programación usando lenguaje de máquina?

¿Qué consideraciones deben ser tenidas en cuenta antes de decidir si un proyecto va a ser programado en lenguaje máquina o en lenguaje de medio o alto nivel?

¿En qué casos resulta mejor enfrentar un proyecto en lenguaje ensamblador que iniciar lo en lenguaje C?

¿Enumere dos aspectos a tener en cuenta al programar un sistema embebido, que sean irrelevantes cuando se programa un PC?

¿Cuál es el tipo de datos más óptimo para el procesamiento en 8 bits?

¿Por qué no existe el concepto de variable volatile en programación en lenguaje ensamblador?

INTRODUCCIÓN

El compilador es un software que resuelve la edición realizada por el diseñador. Es posible dirigir instrucciones al software encargado de la compilación para que tome ciertas acciones antes de la compilación definitiva al microcontrolador; algunas de ellas involucran archivos externos, realizan alguna operación y pueden elegir o descartar una parte de código editado.

Este capítulo describe una a una las directivas de compilación más usadas en los compiladores para sistemas embebidos, pero en especial, las más utilizadas en los compiladores CCS de Microchip™ y Codewarrior ® de Freescale™.

Se trata también el tema de los operadores en C, tanto los básicos aritméticos como los lógicos y orientados a los bits, y en general, todos los que permiten crear la lógica de una expresión actualizando una variable o asignando su valor a una nueva, permitiendo la construcción de los algoritmos de trabajo y la toma de decisiones.

También se discute la sintaxis y cuidados especiales al momento de digitar código que involucra los operadores y se incluyen recetas especiales para que no se tenga dependencia de la precedencia propia del C.

Hacia el final se introduce al lector en un tema de suma importancia, el cual incide en que el código diseñado sea óptimo, como es el concepto de apuntador, su definición, su manejo y su declaración. El completo entendimiento y manejo de este tema posibilita concebir el manejo de datos de una forma muy general.

5.1 DIRECTIVAS MÁS COMUNES DEL PROCESADOR

Las directivas del preprocesador proveen un paso previo a la compilación y permiten realizar ciertas acciones que se resuelven en tiempo de compilación.

Todas las directivas de compilación inician con el signo # en la primera columna del archivo.

5.1.1 Macro o equivalencia #define



Los macros son secuencias cortas de código que se utilizan en muchas partes del programa, no genera código adicional en memoria y permite una modificación sencilla y rápida.

La directiva **#define** define una macro o equivalencia, es similar a la directiva EQU (*Equal*) usada comúnmente en la programación en lenguaje ensamblador, la cual no generará código adicional en memoria ni en tiempo de ejecución. Es usada para declarar equivalencias de variables en valores constantes.

Un macro es una secuencia definida de instrucciones, las cuales son incluidas en el código editado, cada vez que es invocado el nombre del macro.

Los macros son usados ampliamente cuando se requiere usar de forma frecuente una secuencia corta de C, con el objetivo de visualizar el código escrito de forma sencilla, para agilizar y simplificar o simplemente para realizar definiciones o equivalencias.

Un macro ayuda además a que el código quede parametrizable, lo que permitirá modificarlo de forma sencilla y segura.

Su forma de definición es como sigue:

```
#define NOMBRE_MACRO VALOR_O_SECUENCIA_EQUIVALENTE
```

La directiva **#define** deberá iniciar en la primera columna, a continuación va el nombre con el que se invocara la macro **NOMBRE_MACRO**, y seguidamente su respectivo reemplazo **VALOR_O_SECUENCIA_EQUIVALENTE**.



La compilación proporciona el archivo final, pero antes de obtenerlo, se deben conocer las directivas más importantes para que el proceso arroje los resultados esperados.

Aunque no es obligatorio sí es estándar que el **NOMBRE_MACRO** se defina en letras mayúsculas cuando su equivalencia es un valor constante, y combinaciones de mayúsculas y minúsculas (de forma similar a una función), cuando se trata de una secuencia de instrucciones.

Se acostumbra agregar un comentario al final, el cual indica las unidades del valor especificado y una breve descripción.

Ejemplo:

```
#define TEMP_MAX 200 // [=] centígrados, Temperatura Límite
```

```
#define Led1_On() PTC3_PTC3 = 1; DDRC_DDRC3 = 1
```

```
//enciende  
//Led  
  
void main(void){  
    .....  
    .....  
  
    if(temperature > TEMP_MAX){  
        Led1_On();  
    }  
    .....  
    .....  
}
```

Se puede notar su utilidad una vez que se hace la lectura del código en la función, la cual es exactamente equivalente a:

```
void main(void){  
    .....  
    .....  
  
    if(temperature > 200){  
        PTC_PTC3 = 1;  
        DDRC_DDRC3 = 1;  
    }  
    .....  
    .....  
}
```

Este código, aunque genera el mismo resultado final, es más confuso de leer y modificar, y resulta muy peligroso al momento de querer cambiar el valor con el que se compara la variable.

En general, el uso de constantes dentro del código puede generar confusiones al momento de querer variarlo, dado que se tendría que modificar su valor en TODAS las líneas en las cuales se invoca la constante, con el peligro de modificar un valor o una constante que no se debe modificar.

Así, una constante permite tener claridad del código y realizar un reemplazo, en tiempo de compilación, del nombre de la variable por su respectiva equivalencia a la derecha del nombre de la variable.



Una constante permite tener claridad del código y realizar un reemplazo, en tiempo de compilación, del nombre de la variable por su respectiva equivalencia a la derecha del nombre de la variable.



5.1.2 Inclusión de archivo de cabecera #include

Esta directiva es usada para indicar al compilador el nombre de los archivos de cabecera (.H), a los cuales deberá hacer referencia para resolver los tipos de variable, prototipos de funciones y constantes que esté utilizando el archivo (.C).

Esta directiva va incluida al inicio de cada archivo fuente (.C), y seguida del nombre del archivo de cabecera (.H) que se quiere incluir.

Si el nombre del archivo está delimitado por los símbolos < al inicio y > al final, el archivo será buscado en el directorio por defecto del compilador o de las librerías, pero, si el archivo de cabecera está delimitado por “ (comillas dobles) al inicio y ” (comillas dobles) al final, el archivo de cabecera será buscado en el mismo directorio del proyecto actual.

#include <stdio.h> /*incluye el archivo stdio.h, ubicado en el directorio de las librerías*/

#include “leds.h” /*incluye el archivo leds.h ubicado en la dirección del proyecto*/

Una descripción adicional sobre la directiva #include, se verá en el Capítulo 6 Archivos de Cabecera (.H) en la página 255.

5.1.3 Notificación de error al compilar #error



Las notificaciones

de error permiten prohibir ciertas declaraciones del código que obligan (y recuerdan) al programador a realizar las correcciones respectivas antes de ejecutar la compilación.



La directiva `#error` envía un mensaje de error al compilar; proviene de una decisión del programador para evitar que ciertos macros o valores tengan valores inválidos.

En algunos casos se usan para prohibir ciertas declaraciones en el código que obligan a hacer correcciones antes de proceder con la compilación.

Así por ejemplo, se tiene la declaración de una variable como la del Ejemplo 4, en la cual no se permite que la constante NRO_FXS sea mayor que 6, se puede hacer así:

```
#define NRO_FXS 7  
#if NRO_FXS > 6
```

```
#error Valor de NRO_FXS Invalido...
```

```
#endif
```

Al compilar, el preprocesador evaluará la directiva `#if`, que evalúa `NRO_FXS > 6` es verdadero y con esto el código que sigue es incluido y dará como resultado:

The screenshot shows a terminal window with the following output:

```
Error : C4437: Error-directive found: Valor de NRO_FXS Invalido . . .  
main.c line 27  
Error : Compile failed
```

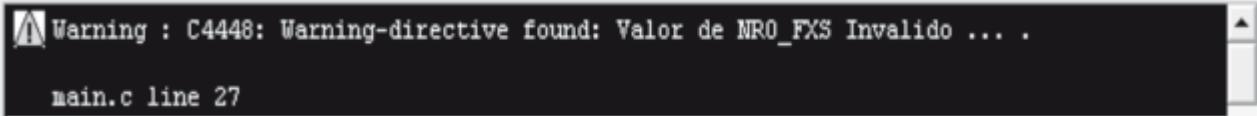
5.1.4 Notificación de precaución al compilar `#warning`

Esta directiva actúa de forma similar al `#error`, solo que envía un mensaje de precaución (*warning*) al momento de compilar, y permitirá la compilación completa (cuando no se tienen errores de sintaxis).

El mensaje enviado en el código:

```
#define NRO_FXS 7  
#if NRO_FXS > 6  
#warning Valor de NRO_FXS Invalido...  
#endif
```

Será:



5.1.5 Directiva #line

La directiva `#line` cambia los contenidos de los nombres predefinidos del ANSI C: `__LINE__` y `__FILE__`. La macro `__LINE__` contiene el número de línea que se está compilando actualmente, mientras que el identificador `__FILE__` es una cadena que contiene el nombre del archivo fuente que se está compilando.

La forma general de `#line` es:

```
#line numero <nombre_de_archivo>
```

Donde:

numero → es una constante entera positiva.

nombre_de_archivo → es opcional y corresponde al nombre de un archivo.

La directiva es usada en códigos que generan a su vez programas en C, depuración y otras aplicaciones especiales, no muy usadas en sistemas embebidos.

Para ilustrar mejor su uso, el siguiente código inicializa el contador de líneas en 100 y la sentencia `printf()` muestra el número 101 por el dispositivo de salida estándar.

```
#include <stdio.h>

#line 100

void main(void){ //.linea 100
    printf("%d\n",__LINE__); //linea 101

    for(;;);
}
```

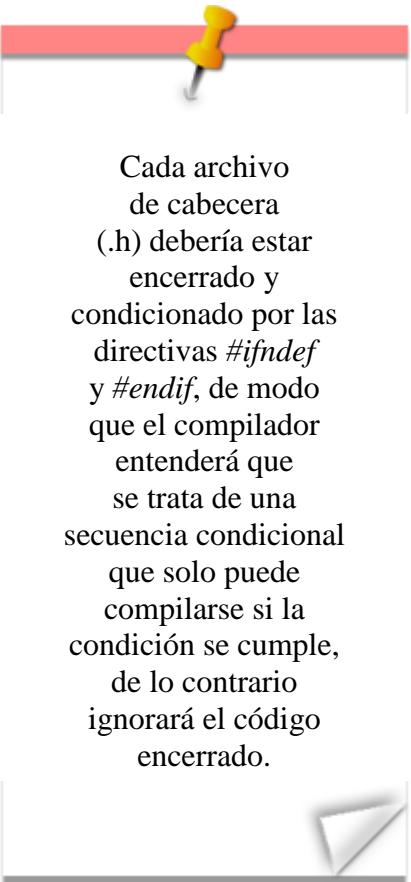


La directiva `#line` es usada en códigos que generan a su vez programas en C, depuración y otras aplicaciones especiales, no muy usadas en sistemas embebidos.



5.1.6 Compilación condicional `#if`, `#elif`, `#else`, `#endif`, `#ifdef` y `#ifndef`

Es posible mediante estas directivas indicar al preprocesador que cierta sección de código está condicionada a compilarse dependiendo de ciertos valores definidos. Esto permite una forma de incluir o no código de forma selectiva sin usar los comentarios `//` y `/* */`.



Cada archivo de cabecera (.h) debería estar encerrado y condicionado por las directivas `#ifndef` y `#endif`, de modo que el compilador entenderá que se trata de una secuencia condicional que solo puede compilarse si la condición se cumple, de lo contrario ignorará el código encerrado.

La directiva `#if` evalúa una constante o expresión, si esta no es cero, las líneas subsecuentes serán incluidas en la compilación hasta encontrar las directivas `#endif` o `#elif` (es como un `else if`) `#else` (sino).

Es usado también en muchas ocasiones para asegurarse que una declaración o un archivo de cabecera son incluidos una sola vez, mediante:

#if !define NOMBRE

#define NOMBRE

/*El código y declaraciones acá presentes solo serán incluidas una única vez*/

#endif

Es una buena práctica que cada archivo de cabecera esté encerrado y condicionado por estas dos directivas `#if` y `#endif`, de tal forma que si varios archivos incluyen el mismo archivo de cabecera, las variables, constantes y macros declarados no sean redeclarados, lo que puede generar un error al momento de compilar.

en la expresión, sino un falso.

El uso de `#ifndef` es similar a `#ifdef`, pero evaluará no un verdadero

Por recomendación si un archivo de cabecera tiene por nombre FILE.H, se puede usar como norma general que el archivo FILE.H tenga la siguiente estructura:

```
***** FILE.H *****
#ifndef FILE__H
#define FILE__H

/*Contenido del archivo de cabecera*/

#endif
```

```
#ifdef VERSION_FULL

/*el código acá solo será incluido si VERSION_FULL es
verdadero*/

#endif

#ifndef VERSION_SMALL

/*el código acá solo será incluido si VERSION_SMALL es falso
o no esta definido en ninguna parte del código*/

#endif
```



5.1.7 Directiva #undef

La directiva **#undef** elimina una definición previa realizada con **#define**. En caso de no existir la definición anterior, no realiza acción.

La forma general es:

```
#undef nombre_macro
```

Así por ejemplo, se pueden manejar declaraciones de macros de forma temporal por sección de código así:

```
#define MATRIZ_A 20
#define MATRIZ_B 10
char array[MATRIZ_A][MATRIZ_B];
#undef MATRIZ_A
```

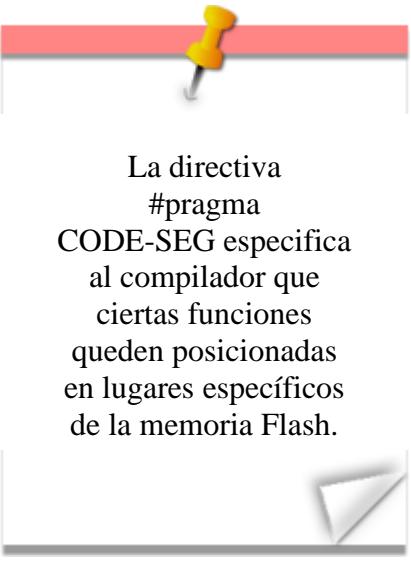
A partir de este momento MATRIZ_A queda indefinida, mientras que MATRIZ_B quedará definida con el valor 10.

Esta definición ayuda a administrar los macros y constantes en las secciones de código que se requieren, o inclusive a declararlos con valores diferentes al original, para el ejemplo anterior MATRIZ_A se puede declarar con un valor diferente al definido inicialmente.

5.1.8 Directiva #pragma

La directiva **#pragma** permite que se den instrucciones al compilador a partir del punto específico en el que se realiza su declaración.

En el compilador existen diferentes **#pragmas**.



La directiva
#pragma
CODE SEG especifica
al compilador que
ciertas funciones
queden posicionadas
en lugares específicos
de la memoria Flash.

#pragma CODE_SEG

Especifica una sección en la cual las funciones estarán posicionadas, dependiendo si se requiere cambiar su posicionamiento a una sección diferente a la determinada por defecto. Es usado para obligar que ciertas funciones queden posicionadas en una sección de memoria de programa (Flash) específica.

Su sintaxis es:

#pragma CODE_SEG (<modif><name>|DEFAULT)

Donde:

<modif> → es alguna de las siguientes constantes: __DIRECT_SEG,
, __NEAR_SEG, __CODE_SEG, __FAR_SEG.

#pragma CONST_SEG

Especifica una sección en la que las constantes y variables tipo const de una sección específica serán posicionadas.

Su sintaxis es:

#pragma CONST_SEG (<modif><name>|DEFAULT)

Donde:

<modif> → es uno de los siguientes: __SHORT_SEG, __DIRECT_SEG, __NEAR_SEG,
, __CODE_SEG, __FAR_SEG, dependiendo de la sección seleccionada.

Las diferentes secciones están definidas en el archivo *linker* (.PRM) del proyecto.

Así por ejemplo, se pueden hacer las siguientes declaraciones:

// Declaraciones en el Archivo de cabecera

#pragma CONST_SEG _MY_SEG_CONSTS

extern const int contador_max;

// ésta variable quedará localizada en la sección _MY_SEG_CONST

#pragma CONSTS_SEG DEFAULT

// Retorna el posicionamiento de las constantes a la sección por defecto.

#pragma CREATE_ASM_LISTING

Este **#pragma** habilita o no la generación de los objetos que se encuentran a continuación. Por defecto está OFF, una vez encendido ON genera símbolos para que estos puedan ser accesados desde el código en ensamblador.

Su sintaxis es:

```
#pragma CREATE_ASM_LISTING (ON|OFF)
```

Así por ejemplo:

```
#pragma CREATE_ASM_LISTING ON
```

```
extern int i; // la variable i, podrá ser accesada desde el código en ensamblador
```

```
#pragma CREATE_ASM_LISTING OFF
```

```
extern int j; // la variable j solo podrá ser accesada desde el código en C
```

#pragma DATA_SEG

Permite localizar las variables (datos) declaradas en el código en secciones específicas de la RAM. La directiva **#pragma DATA_SEG** es usada para localizar variable de forma selectiva en la memoria, a fin de no usar los modificadores *near* o *far* de los modificadores de variables.

También permite que ciertas variables estén localizadas en una sección definida por el programador en el archivo **linker (.PRM)**.

Su sintaxis es:

```
#pragma DATA_SEG (<modif.> <name>|DEFAULT)
```

Donde:

<modif> ◊ toma los valores de: _SHORT_SEG, _DIRECT_SEG, _NEAR_SEG, _CODE_SEG, _FAR_SEG, dependiendo de la sección a cambiar.

Así por ejemplo:

```
#pragma DATA_SEG __SHORT_SEG MY_ZERO_PAGE
```

```
char i,j,k;
```

```
/* Estas variables estarán localizadas en la zona directa de RAM (0x0000 a 0x00FF) */
```

```
#pragma DATA_SEG DEFAULT
```

#pragma INLINE #pragma NO_INLINE

Esta directiva le indica al compilador que la siguiente función una vez invocada deberá ser incluida en línea (similar al macro), sin realizar llamado a subrutina.

Así por ejemplo:

```
int i;  
  
#pragma INLINE  
  
static void Funcion(void){  
  
    i = 40;  
  
}  
  
void main(void){  
  
    Funcion(); /*no se realiza JSR o BSR, en su lugar se reemplaza por i= 40;*/  
  
}
```

De forma similar existe la declaración NO_INLINE donde se especifica de forma explícita que la siguiente función no se debe insertar en línea, sino en su lugar hacer el llamado a la función con las instrucciones BSR o JSR.

#pragma TRAP_PROC

Esta directiva indica que la función a continuación es una función de interrupción y no una subrutina, de forma que antepone y retorna de forma especial.

Esta directiva solo tiene efecto en la función que sigue a su definición, por lo que las funciones que continúan en el código no serán afectadas por esta directiva, de tal forma que cada función de interrupción deberá estar antecedida por su respectiva directiva.

Así por ejemplo:

```
#pragma TRAP_PROC  
  
void Funcion_ISR(void){  
  
    .... // código de la interrupción  
  
}
```

Indicará al compilador que la función **Funcion_ISR()** tiene tratamiento de función de interrupción, estas funciones siempre tienen el prototipo void F(void), debido a que el código externo no está realizando un llamado explícito a la función, y por tal motivo no lleva argumentos, ni tampoco retorna valores.

5.2 MEDIDA EN BTES DE EXPRESIONES “sizeof”

El operador “*sizeof*” entrega el **TAMAÑO** en bytes de una variable o la expresión que se encuentre entre los () que siguen al “*sizeof*”.

Este operador resuelve su resultado en **tiempo de compilación**, ya que no tiene en cuenta el valor de su argumento, sino su longitud, la cual está perfectamente definida desde su declaración.

Ahora, es importante considerar que este tamaño puede variar de un compilador a otro, por lo que no todos los tipos de variables tienen igual longitud y depende mucho de la máquina para la cual se desea realizar la compilación. Este problema es serio, si luego se quiere compilar el programa con un compilador distinto del original.

Para salvar esta dificultad y mantener la portabilidad, es conveniente que cada vez que se requiera referirse al **TAMAÑO** en bytes de las variables, se lo haga mediante el operador llamado “*sizeof*”, que calcula sus requerimientos de almacenaje.



El tamaño
de bytes de
una expresión

puede variar de un
compilador a otro, de
modo que el tamaño
final dependerá sobre
todo de la máquina
para la cual se hace
la compilación.



```
tamano = sizeof(<expresion>);  
  
// tamano tendrá el número de bytes de <expresion>
```

Está también permitido el uso de “*sizeof*” para los diferentes tipos de variable, es decir:

`sizeof(int)` → dará como resultado **2**

`sizeof(char)` → dará como resultado **2**

`sizeof(long)` → dará como resultado **4**

`static unsigned char vble1;`

`sizeof(vble1)` → dará como resultado **1**

Resulta muy útil también al manejar apunadores, estructuras, uniones y cualquier operación en la que se dependa del tamaño de las variables.

5.3 DEFINICIONES DE TIPO “*typedef*”

Tal y como se describieron, existen en C los tipos de datos básicos **char**, **int**, **long** y **float**, antecedidos de sus respectivos modificadores.

Sin embargo, el programador podrá realizar definiciones de sus propios tipos de datos que se basan en los ya existentes por el ANSI C.

La forma en la que se deberá hacer la declaración es la siguiente:

typedef tipo_ANSI	nuevo_tipo;
--------------------------	--------------------



En donde:

tipo_ANSI: corresponde al tipo ANSI acompañado de sus modificadores.

nuevo _ tipo : es el nombre del tipo que el usuario quiere definir

Existen dos razones fundamentales para hacerlo:

Mayor sencillez a la hora de declarar una variable: en este caso **nuevo_tipo** tiene una forma más sencilla de invocar el tipo de una variable a definirse.

Definir un tipo específico usado en el proyecto.

Ejemplos de definición de tipo: **typedef**

Considérese inicialmente que en el código se requiere declarar frecuentemente variables de tipo **volatile unsigned char**.

La declaraciones serían así:

```
volatile unsigned char vble1;
```

...

```
volatile unsigned char vble2;
```

...

Y de forma similar cuando se requieran más variables.

Se puede entonces usar **typedef** para simplificar las declaraciones de este tipo de variables así:

```
typedef volatile unsigned char vuchar;
```

y realizar las declaraciones de forma más corta así: **vuchar vble1;**

...

```
vuchar vble2;
```

Lo cual resulta más corto de editar, y sencillo de visualizar.

Considérese ahora que se tiene un proyecto en el cual existen 2 sensores que miden 2 variables externas: temperatura y presión, para cada una de ellas se define un tipo específico y para ello se consideran las siguientes definiciones de tipo:

```
// [=] en centígrados, con posibilidad de
```

```
typedef volatile signed char temperatura; // valores positivos hasta +127 y
```

```
// negativos hasta -128.
```

```
typedef volatile unsigned char presion; // [=] en PSI, solo toma valores positivos.
```

De esta forma, una vez se necesite declarar una variable de tipo **temperatura**, se hará de forma sencilla así:

```
temperatura temp1,temp2; //se declaran 2 variables de tipo temperatura
```



La definición **typedef** es de gran ayuda, porque designa una variable que más adelante podría cambiar de valor y ya no tendrían que cambiarse todas las declaraciones donde aparezca sino solamente en su declaración **typedef**.

Y la declaración de un par de variables de tipo **presión** así: **presion pres1,pres2;**

La declaración resulta más sencilla y el tipo resulta ser más congruente con la realidad del proyecto.

La ventaja en mantenimiento de software consiste en que si en un futuro la variable **temperatura** necesitara ser cambiada de dimensión o de definición, no se requeriría hacer el cambio en cada una de las declaraciones y en cada archivo .C en el cual se encuentre su declaración, sino por el contrario solo en la definición de su tipo **typedef**.

Imagínese que el proyecto creció o se requiere más precisión, y ahora la variable **temperatura** no puede almacenarse en un tipo **char**, sino que requiere de un valor entero: int.

Para este caso solo se requiere cambiar su dimensión en la definición

de tipo así:



typedef volatile signed int temperatura; /* [=] en centígrados, con posibilidad de valores positivos hasta +32767 y negativos hasta -32768 */

Evitando de esta forma la necesidad de modificarla en cada uno de los archivos y definiciones del proyecto.

Sin el **typedef** se tendría que ir por cada una de las definiciones de la variable modificando **signed char** por **signed int**, con el peligro adicional de cambiar la declaración de otras variables que no se espera cambiar.

Es importante anotar que luego se debe compilar todo el proyecto para que el cambio se haga efectivo en cada una de las declaraciones de la variable, después de esto, todas las variables realizarán su nueva declaración a la nueva dimensión.

5.4 OPERADORES ARITMÉTICOS

Un operador aritmético es un símbolo que indica al compilador que lleve a cabo manipulaciones matemáticas.

Los operadores aritméticos en C se resumen en la Tabla 5.1.

Operador	Operación Realizada
-	Resta o substracción
+	Suma o adición
*	Multiplicación o producto
/	Cociente entero de la división
%	Residuo entero de la división
--	Decremento en 1
++	Incremento en 1

- Resta

El operador resta opera sobre dos variables o expresiones y dará como resultado la substracción de ellos. Es responsabilidad del usuario conservar o no el signo del resultado.

```
vble3 = vble1 - vble2;
```

+ Suma

El operador suma, realiza la operación sobre dos variables o expresiones. El programador deberá cuidar de posicionar el resultado en la variable adecuada, en caso de existir sobreflujo por ejemplo.

```
unsigned char vble1,vble2;
```

```
unsigned int resultado;
```

```
resultado = (unsigned int)vble1 + (unsigned int)vble2;
```

*: Multiplicación

Este operador realiza la multiplicación de dos expresiones, variables o constantes, entregando como resultado un nuevo dato y conservando el signo, según la ley de multiplicación.

```
unsigned char vble1,vble2;
```

```
unsigned long resultadoMul;
```

```
resultadoMul = (unsigned long)vble1*(unsigned long)vble2;
```

/ Cociente de división



Los principales operadores aritméticos son:
- Resta, + Suma,
* Multiplicación,
/ Cociente de división, % Residuo de división, ++
Incremento en 1 y --
Decremento en 1.

El operador aplicado sobre un dato entero o carácter, entregará el cociente entero de la división, cualquier resto es truncado.

Así que si se opera: 10/3 el resultado entregado será 3.

% Residuo de división



El operador aplicado sobre dos expresiones, constantes o variables enteras (nunca flotantes), entregará como resultado el residuo o módulo de la división.

Así por ejemplo:

```
int x,y,result;  
x = 10;  
y = 3;  
result = x%y; // result quedará con el dato 1  
x = 1;  
y = 2;  
result = x%y; // result quedará con el dato 1
```

++ Incremento en 1

Este operador adiciona un 1 a la variable sobre la cual está aplicado, que es equivalente al operador suma con 1 así:

vble = vble + 1;

es equivalente a:

vble++;

y equivalente a:

++vble;

-- Decremento en 1

El operador -- resta 1 a la variable sobre el cual esta aplicado, que es equivalente al operador resta con 1 así:

vble = vble - 1;

Es equivalente a:

vble--;

y equivalente a:

--vble;

Existe una diferencia cuando los operadores -- y ++ se utilizan en una expresión. Cuando alguno de ellos precede a su operando, el C lleva a cabo una operación de incremento o decremento antes de usar el valor del operando (pre-incremento o pre-decremento), mientras que si el operador esta después de la variable, el C utiliza su valor antes de incrementarlo o decrementarlo (pos-incremento o pos-decremento).

Considérese lo siguiente:

x = 5;

y = x++;

Al final la variable y tomará el valor de 5 mientras que x tomará el valor de 6.

Mientras que:

x = 5;

y = ++x;

La variable y tomara el valor de 6, lo mismo que la variable x.



5.5 OPERADORES RELACIONALES

Todas las operaciones relacionales dan sólo dos posibles resultados: VERDADERO ó FALSO. En el lenguaje C, FALSO queda representado por un valor entero nulo (cero) y VERDADERO por cualquier número distinto de cero.

En la Tabla 5.2 se encuentra la descripción de los operadores relacionales del C:

SÍMBOLO	DESCRIPCIÓN	EJEMPLO
<	menor que ...	if(a < b)
>	mayor que ...	if(a > b)
< =	menor o igual que ...	if(a < = b)
> =	mayor o igual que ...	if(a >= b)
= =	Es exactamente igual que...	if(a = = b)
! =	diferente que ...	if(a != b)

5.5.1 Operadores de Comparación Cuantitativa: <, > ,<=,>=

Dentro de una expresión los operadores < “menor que”, > “mayor que”, <= “menor o igual que” y >= “mayor o igual que”, son usados para comparar el valor de dos (2) cantidades, arrojando como resultado un VERDADERO o un FALSO dependiendo del resultado de la comparación de ambos valores. Se usan también en varias sentencias de C como el **while**, el **do{}while** y el **for**, para generar iteraciones o sacar el programa de la sentencia bajo una condición o valor determinado de una variable.

Cuando se comparan dos variables tipo **char** el resultado de la operación dependerá de la comparación de los valores ASCII de los caracteres contenidos en ellas. Así el carácter ‘a’ (ASCII 97) será mayor que el ‘A’ (ASCII 65) ó que el ‘9’ (ASCII 57).

5.5.2 Operador de igualdad: ==, !=

Estos operadores realizan la comparación de sus dos (2) valores a lado y lado de ellos y arrojan como resultado VERDADERO o FALSO, dependiendo del resultado de la comparación.]

Son útiles cuando se intenta verificar la igualdad de 2 valores y con ello tomar alguna decisión dentro de sentencias while, do{} while, if .. else, for.

Uno de los errores más comunes es confundir el operador relacional “es exactamente igual que” (==) con el de asignación (=). La expresión (a=b) copia el valor de b en a, mientras que (a == b) retorna un cero, si a es distinto de b ó un número distinto de cero si son iguales.



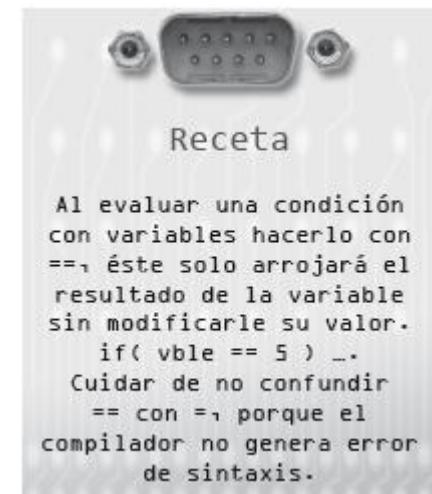
El operador de igualdad realiza la comparación de sus dos (2) valores a lado y lado de ellos y arrojan como resultado VERDADERO o FALSO.



5.6 OPERADORES LÓGICOS BOOLEANOS

Los operadores lógicos booleanos, y de forma similar a los relacionales, entregan solo dos (2) valores: VERDADERO o FALSO y se basan en las reglas de la álgebra booleana.

```
result = <expresión1> OPERADOR_BOOL <expresión2>;
```



Los operadores lógicos booleanos se resumen en la Tabla 5.3 y se describen a continuación:



Operadores lógicos booleanos del C.

SÍMBOLO	DESCRIPCIÓN	EJEMPLO
	OR	if(a b)
&&	AND	if(a && b)
!	NOT	if(!a)

5.6.1 El operador || (OR)

Este operador corresponde al operador OR BOOLEANO o de expresión completa. Arroja solo dos (2) resultados --> FALSO (F) o VERDADERO (V), de acuerdo con la tabla OR definida así:



Operador OR.

Operando 1	Operando 2	Resultado
V	V	V
F	V	V
V	F	V
F	F	F

Útil para conectar dos o más expresiones y tomar una decisión cuando alguna de ellas se cumpla o sea VERDADERA.

Es común encontrarla en sentencias de control **if{}else{}, while{} y do{}while()**, dentro de la condición de evaluación.

Dentro de la secuencia **if(<condicion>){}else{}{}** por ejemplo, puede generar el siguiente comportamiento al código:

```
if(<expr[1]> || <expr[2]> || <expr[3]> ... || <expr[n]>){
    BLOQUE DE CODIGO VERDADERO
}else{
    BLOQUE DE CODIGO FALSO
}
```

Si alguna de las **<expr[i]>** es VERDADERA se ejecutará el BLOQUE DE CÓDIGO VERDADERO, y solo en el caso que TODAS las **<exp[i]>** sean falsas se ejecutará el BLOQUE DE CÓDIGO FALSO.

5.6.2 El operador && (AND)

Este operador corresponde al operador AND BOOLEANO o de expresión completa. Arroja solo dos (2) resultados --> **FALSO (F)** o **VERDADERO (V)**, de acuerdo con la tabla AND definida así:



Operador && (AND).

Operando 1	Operando 2	Resultado
V	V	V
F	V	F
V	F	F
F	F	F

Este operador es usado para conectar dos o más expresiones, y tomar una decisión cuando estas sean ciertas.

Es costumbre encontrarla o usarla como evaluación de la condición de la sentencia `if(<condicion>?){ }else{ }`, siendo esta una composición de varias expresiones, así:

```
if(<expr[1]> && <expr[2]> && <expr[3]> ... && <expr[n]>){  
    BLOQUE DE CODIGO VERDADERO  
}else{  
    BLOQUE DE CODIGO FALSO  
}
```

En este caso solo se ejecutará el BLOQUE DE CODIGO VERDADERO, si TODAS las expresiones `<expr[i]>` son verdaderas, y con solo una de ellas que sea falsa, se ejecutará el BLOQUE DE CODIGO FALSO.

5.6.3 El operador ! (NOT)

Es un operador unitario, el cual realiza su acción sobre una sola variable, registro o expresión. Realiza la negación completa de su argumento ubicado a la derecha del operador:

resultado = !<expresion>;

Si la expresión es VERDADERA, el resultado luego de la operación será FALSO, mientras que si la expresión es FALSA, el resultado será negado y convertido a VERDADERO.

Es útil en secuencias de control donde se requiere la condición contraria a su estado booleano actual o ajustar la lógica negada que tenga un pin del microcontrolador.

Así por ejemplo, en la secuencia if(<condicion>?){ }else{ }, es usual tener expresiones de la siguiente forma:

```
if(!Tecla_Presionada()){
    B LOQUE DE CODIGO VERDADERO
}else{
    BLOQUE DE CODIGO FALSO
}
```

O de forma común esta en expresiones como:

```
if( (caracterRecibido) && (!TimeOut) ){
    BLOQUE DE CODIGO VERDADERO
}else{
    BLOQUE DE CODIGO FALSO
}
```

5.7 OPERADORES ORIENTADOS A BIT

El lenguaje C, y a diferencia de otros lenguajes de programación, permite realizar la operación de expresiones a nivel de bit, los cuales usan varios operadores que se resumen en la Tabla 5.6 y se describen a continuación:

Operadores orientados a BIT en C.			
TABLA 5.6	OPERADOR_BIT	DESCRIPCIÓN	EJEMPLO
		OR bit	a = a 0x80;
	&	AND bit	if(a & 0x01)
	~	NOT bit	a = a & (~0x80)
	^	XOR bit	a ^= 0x80;

Los operadores orientados a bit a diferencia de los booleanos que realizan operación sobre una expresión completa, la hacen a nivel de bits, operando las dos expresiones a lado y lado de su expresión bit con bit correspondiente de la expresión, así:

```
result = <expresión1> OPERADOR_BIT <expresión2>;
```

En donde:

result[0] → bit 0 (menor peso) = <expresión1>[0] OPERADOR_BIT <expresión2>[0]

result[1] → bit 1 = <expresión1>[1] OPERADOR_BIT <expresión2>[1]

result[2] → bit 2 = <expresión1>[2] OPERADOR_BIT <expresión2>[2]

....

result[n] → bit n (mayor peso) = <expresión1>[n] OPERADOR_BIT <expresión2>[n]

Donde n será 7 en caso de operaciones de tipo **char** ó 15 en operaciones de tipo **int**, y 31 en operaciones de tipo **long**.

Las operaciones a nivel de bits no se pueden usar sobre los tipos **float**, **double**, **long double** u otros tipos más complejos.

Las operaciones sobre bits son frecuentes en aplicaciones de controladores de dispositivos, tales como programas de modem, rutinas de archivo de disco, impresoras, entre otros, debido a que permiten enmascarar ciertos bits, como el de paridad o de estados.

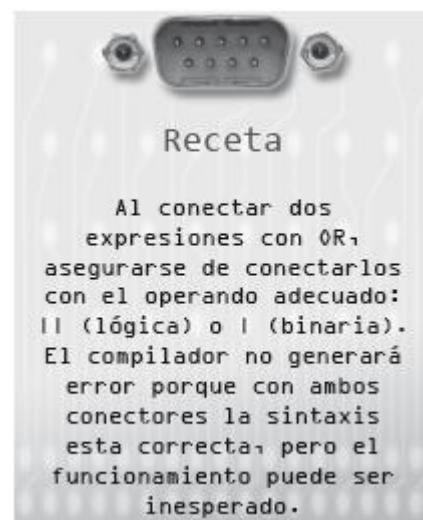
5.7.1 El operador | (OR Bit a Bit)

Denominado el operador OR binario, realiza la operación OR BIT a BIT de los operandos a su derecha y a su izquierda, y arroja el valor resultante de la operación, según la tabla de verdad OR de bits definida así:

Operador OR Bit a BIT.

TABLA 5.7

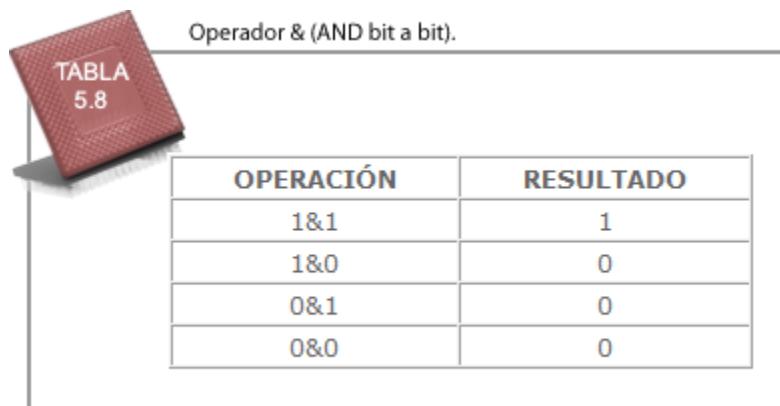
OPERACIÓN	RESULTADO
1 1	1
1 0	1
0 1	1
0 0	0



Para el caso de los sistemas microcontrolados es ampliamente usado para encender (asignar UNO lógico) a uno a varios bits de un puerto, registro interno o variable en RAM, dejando los bits señalados en CERO sin modificarlos.

5.7.2 El operador & (AND bit a bit)

Llamado el operador AND Binario, realiza la operación AND BIT a BIT de los operandos a su derecha y a su izquierda y arroja este valor como resultado de la operación, según la tabla de verdad AND de bits:



Operador & (AND bit a bit).	
OPERACIÓN	RESULTADO
1&1	1
1&0	0
0&1	0
0&0	0

En un sistema embebido es ampliamente usado para borrar (asignar CERO lógico) a uno o varios bits de un puerto, de un registro interno o de una variable en RAM, dejando los demás bits intactos.

Así por ejemplo, si se quiere apagar el BIT menos significativo del PUERTO_B, se hará mediante la siguiente línea:

PUERTO_B = PUERTO_B & 0b11111110;

De igual forma se puede querer apagar el bit de mayor peso y el de menor peso, de la siguiente forma:

PUERTO_B = PUERTO_B & 0b01111110;

Nota Importante 1: no se debe confundir el operador & (AND) con el operador unitario &, este último da como resultado la dirección de un dato.

Nota Importante 2: es de vital importancia tener un especial cuidado al momento de usar algunos de los operadores AND y OR y recordar si el uso será binario o booleano, ya que sintácticamente puede estar correcto usar & o &&, lo mismo | como ||, pero puede no ser lo que el programador quiere realizar. Esto sucede porque el C no verifica los tipos de datos que se operan, si son valores o son expresiones booleanas.

Vea el siguiente ejemplo:

```
if( (vble1 && vble2 ){
```

```
BLOQUE DE CODIGO VERDADERO  
}
```

y

```
if( (vble1 & vble2 ){  
    BLOQUE DE CODIGO VERDADERO  
}
```

Y supóngase los siguientes valores:

vble1 --> 0x55

vble2 --> 0xAA

Para el primer caso la expresión a resolver es (0x55 && 0xAA), esta operación dará como resultado VERDADERO, ya que 0x55 es VERDADERO y 0xAA es VERDADERO, resultando en la ejecución del BLOQUE DE CODIGO VERDADERO de la secuencia if{}.

Sin embargo, al unirlos con el operador &, la operación se realizará a nivel de bits así:

0x55 --> 0b01010101

0xAA --> 0b10101010

& --> 0b00000000 ----> FALSO

Resultando que NO se ejecute el BLOQUE DE CODIGO VERDADERO dentro de la secuencia if{ }

5.7.3 El operador ~ (NOT bit a bit)

Es un operador unitario NOT o complemento a uno, opera bit a bit una expresión, variable, puerto, registro o constante y entrega como resultado su complemento, si el bit es 1, entregará un 0 y si es un 0, entregará un 1.

Es útil en expresiones donde se requieren operaciones a nivel de bits que involucran el complemento en bits.

En los sistemas embebidos es útil para hacer referencia a BITS de forma sencilla, así por ejemplo, se quiere de forma similar encender y apagar bits, como en el Ejemplo anterior, pero se requiere ahora borrar los BITS de mayor y menor peso del PUERTO_C del microcontrolador, retomando el ejemplo la línea es:



PUERTO_C = PUERTO_C & 0b01111110;

Con la ayuda del operador unitario ~ sería de la siguiente forma:

PUERTO_C = PUERTO_C & ~(0x80 | 0x01);

O de forma simplificada:

PUERTO_C &= ~(0x80 | 0x01);

Esta última expresión es mucho más amigable para aquel que lee de nuevo el código o quiere modificarlo, para realizar acción sobre otros bits.

Es importante anotar que cuando el operador actúa sobre una constante el resultado lo puede desarrollar en tiempo de compilación, por lo que es claro que aunque este último ejemplo involucre mas operaciones que el ejemplo inicial, no tomará mas tiempo su ejecución, de hecho es claro que $\sim(0x80 | 0x01)$ es 0x7E (0b01111110) y puede ser perfectamente resuelto al momento de compilar.

5.7.4 El operador ^ (OR exclusiva o XOR)

El operador ^ realiza la operación bit a bit sobre las 2 expresiones y retorna el resultado del bit correspondiente, según la tabla de verdad XOR así:

El operador ^ (OR exclusiva o XOR).

TABLA 5.9

OPERACIÓN	RESULTADO
1^1	0
1^0	1
0^1	1
0^0	0

Este operador es muy usado en algoritmos de encriptación CRC (código de redundancia cíclica) debido a que muchos están basados en esta operación por ser mucho más segura que el sencillo CHECKSUM5.

A nivel de hardware del microcontrolador se usa para realizar acciones sobre los pines debido a que puede reducir código al querer solo cambiar el estado de un pin por ejemplo.

Suponga que se necesita cambiar el estado del bit de mayor peso del **PUERTO_B** (PTB7) del microcontrolador cada vez que se invoque la función TogglePTB7, usando la lógica anterior la rutina se vería así:

```
void TogglePTB7(void){
```

```

if(PUERTO_B & 0x80){ // verifica si el bit7 esta en "1"
    PUERTO_B &= ~0x80; // apaga el bit7
}else{
    PUERTO_B |= 0x80; //enciende el bit7
}
}

```

Mientras que con el operador ^ se vería de la siguiente forma:

```

void TogglePTB7 (void){

    PUERTO_B ^=0x80;

}

```

Con la ventaja de poder realizar acción sobre varios de los pines, únicamente usando la máscara adecuada.

EJEMPLO No. 11

Uso de operadores lógicos AND y OR

Objetivo:

Escribir un programa en C que realice el encendido y apagado periódico de la salida OUT-1 de forma visible, modificar la frecuencia de encendido mediante las teclas INPUT-1 para incrementar y la tecla INPUT-2 para decrementar.

Adicionar una rutina de verificación de error en la cual activa la salida BUZZ si ambas teclas son presionadas al tiempo.

Solución:

```

***** Ejemplo 11 *****
// Uso de operadores lógicos AND y OR
// Fecha: Abril 1,2009
// Asunto: Encendido y apagado periódico
// de LED OUT-1 con condiciones.
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// o PIC-Link (2008-12-15).
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#define __FREESCALE_
#undef __PIC__
#endif __FREESCALE_

```

```

#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP
#define OUT_1_On() PTC_PTC3 = 1;DDRC_DDRC3 = 1
#define OUT_1_Off() PTC_PTC3 = 0;DDRC_DDRC3 = 1
#define Key1_Press() !PTD_PTD1
#define Key2_Press() !PTD_PTD2
#define Buzzer_On() PTD_PTD0 = 1;DDRD_DDRD0 = 1
#define Buzzer_Off() PTD_PTD0 = 0;DDRD_DDRD0 = 1
#endif
#ifndef __PIC_
#include "operadorAND_OR.h"

#define OUT_1_On() output_high(PIN_B4)
#define OUT_1_Off() output_low(PIN_B4)
#define Key1_Press() !input(PIN_B0)
#define Key2_Press() !input(PIN_B1)
#define Buzzer_On() output_high(PIN_B2)
#define Buzzer_Off() output_low(PIN_B2)
#endif
unsigned long cntDelay=25000;
#define DELTA_DELAY 1000
#define CNT_DELAY_MAX 60000
void Key_Read(void){ //Función de lectura de teclado
    if(Key1_Press()&&(cntDelay > DELTA_DELAY)){ cntDelay -= DELTA_DELAY; }
    if(Key2_Press()&&((cntDelay+DELTA_DELAY)<CNT_DELAY_MAX)){
        cntDelay+= DELTA_DELAY;
    }
    if(Key1_Press() && Key2_Press()){ Buzzer_On(); }else{ Buzzer_Off(); }
}
void Delay(void){ //Función de retardo de conteos de cntDelay
unsigned long i;
    i=cntDelay;
    while(i){ i--; }
}
void Out1_Blink(void){ //Función de encendido y apagado de salida OUT-1
    OUT_1_On();
    Delay();
    OUT_1_Off();
    Delay();
}
void Mcu_Init(void){ //Función de inicialización del microcontrolador
#ifndef __FREESCALE_
    Disable_COP();
#endif
#ifndef __PIC_
    Mcu_Init_PIC();
#endif
}

```

```

}

void main(void) { //Función principal
    Mcu_Init();
    for(;;) {
        Key_Read();
        Out1_Blink();
    } /* loop forever */
}

```

Discusión:

Inicialmente el programa declara los macros necesarios para encender y apagar el LED **OUT-1** y el Buzzer **BUZZ**, y los macros para realizar la lectura de **INPUT-1** y de **INPUT-2**.

Se declara una variable global llamada **cntDelay**, que llevará el valor del conteo a realizar en cada ciclo de retardo **Delay()**, la función **Key_Read()** la cual suma o resta de forma acotada, a la variable **cntDelay**, para actualizar el nuevo valor de retardo.

La función principal **main()** por su parte solo realiza llamado de las funciones **Key_Read()** y **Out1_Blink()**, las cuales efectúan lectura de teclado y ejecución del encendido y apagado del led **OUT-1**.

Al usar los operadores lógicos AND (**&&**) y OR (**||**), y con el objetivo de tener mayor claridad en las expresiones, se recomienda que ambos lados del operador tengan los respectivos paréntesis que obligan la precedencia que requiere el programador.

En el caso del ejemplo la línea:

(**Key1_Press()** **&&** (**cntDelay > DELTA_DELAY**)).

5 Checksum: Suma aritmética.

5.8 LOS OPERADORES **>>Y<< (DESPLAZAMIENTO)**

Los operadores de desplazamiento de bits **>>** y **<<**, realizan un movimiento de los bits de una variable ya sea a la derecha **>>** o a la izquierda **<<** según se especifique. La forma general de una secuencia de desplazamiento es:

```
resultado = vble >> nro_bits_desplazar;  
resultado = vble << nro_bits_desplazar;
```

Donde “vble” es la variable sobre la que se va a operar el desplazamiento, y “nro_bits_desplazar” corresponde al número de pasos de desplazamientos que se van a aplicar sobre la variable. El resultado puede ser posicionado sobre otra variable sin modificar la operada “vble” o sobre la misma, en cuyo caso se puede usar la abreviación usada en los operadores aritméticos $>>=$ o $<<=$.

Así por ejemplo:

```
c = a << b;
```

Implica asignarle a la variable c, el valor de a con sus bits corridos a la izquierda en b lugares; los bits que van “saliendo” por la izquierda, se desechan y los bits que van quedando libres a la derecha se completan con cero.

Se procede de la misma manera para el corrimiento a la derecha $>>$.

De forma diferente a algunas instrucciones en lenguaje ensamblador, los operadores de desplazamiento NO son una rotación, o mejor los bits que salen por un extremo no se introducen por el otro lado. Los bits que salen por un lado, se pierden a medida que se introducen los ceros por el lado opuesto. Sin embargo un desplazamiento a la derecha de un número negativo introduce unos con el ánimo de conservar el signo.

Los operadores de desplazamiento de bits pueden ser muy útiles y eficientes en operaciones de multiplicación o división por 2 (base binaria), recuerde que a nivel binario, multiplicar por 2 equivale a desplazar el valor a la izquierda, mientras que dividir por 2 equivale a desplazar a la derecha.

También son útiles al querer usar palabras de forma separada, por nibbles6, bytes de forma separada, o al desear enviar de forma serial el contenido de una palabra, o recibir una información serial y asignarla a una palabra completa proveniente de una memoria externa o un conversor D/A.

EJEMPLO No. 12

Uso de operadores desplazamiento

Objetivo:

Escribir un código en C que toma el valor de una variable long (4 bytes) e invierta el orden de los bytes de mayor y menor peso.



Los operadores de desplazamiento

NO son una rotación, porque los bits que salen por un extremo no se introducen por el otro lado sino que se pierden a medida que se introducen los ceros por el lado opuesto.



Para el long **byte4:byte3:byte2:byte1**, el dato resultante después de ejecutar la rutina deberá quedar en el orden: **byte1:byte2:byte3:byte4**.

El nuevo valor deberá ser mostrado por nibbles (iniciando por los de mayor peso) por cuatro de los pines microcontrolador, cada vez que la tecla INPUT-1 es presionada.

Solución:

```
***** Ejemplo 12 *****
// Uso de operadores desplazamiento <> <>
// Fecha: Feb 4,2009
// Asunto: Inversión de bits de un long y
// presentación por nibbles.
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// Freescale™ y PIC-Link para Microchip™.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#define __FREESCALE_
#ifndef __PIC_
#endif __FREESCALE_
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP
#define Key1_Press() (!PTD_PTD1)
#endif
unsigned long vbleOriginal;
unsigned long vbleFinal;
volatile unsigned char portVble;
volatile unsigned char portBck; // valor temporal del puerto
#define ActualizaPuerto() portBck &= ~0xF0; portBck |= portVble; PTB = portBck;
DDRB |= 0xF0
#ifndef __PIC_
#include "desplazamiento.h"
#define Key1_Press() (!input(PIN_B0))
static unsigned int32 vbleOriginal;
unsigned int32 vbleFinal;
volatile unsigned char portVble;
volatile unsigned char portBck; // valor temporal del puerto
#define ActualizaPuerto() portBck &= 0x0F; portBck |= portVble; portBck = portBck>>4;
output_a(portBck)
#endif
unsigned char rotaBit=31;
void Mcu_Init(void){
#ifndef __FREESCALE_
Disable_COP();
#endif
}
}
```

```

void main(void) { //Función principal
#ifndef __FREESCALE__
unsigned long vbleTest;
#endif
#ifndef __PIC__
unsigned int32 vbleTest;
#endif
Mcu_Init();
vbleOriginal = 0xFE45EDB5;
vbleTest = 0;
vbleTest |= (vbleOriginal >> 24);
vbleTest |= (vbleOriginal << 24);
vbleTest |= (vbleOriginal & 0x00FF0000) >> 8;
vbleTest |= (vbleOriginal & 0x0000FF00) << 8;
vbleFinal = vbleTest;
for(;;) {
    portVble = 0;
    while(!Key1_Press());
    if((vbleFinal >> rotaBit) & 0x01) portVble |= 0x80;
    rotaBit--;
    if((vbleFinal >> rotaBit) & 0x01) portVble |= 0x40;
    rotaBit--;
    if((vbleFinal >> rotaBit) & 0x01) portVble |= 0x20;
    rotaBit--;
    if((vbleFinal >> rotaBit) & 0x01) portVble |= 0x10;
    ActualizaPuerto();
    while(Key1_Press());
    if(rotaBit) rotaBit--; else rotaBit = 31;
} /* loop forever */
}

```

Discusión:

Se toma la variable original **vbleOriginal**, se rota a la derecha y a la izquierda para invertir los bytes alto y bajo.

Luego se rotan 8 lugares a la derecha y a la izquierda para invertir los bytes intermedios. Con esto se resuelve la primera parte del problema en la que la **vbleOriginal** quedará con sus 4 bytes invertidos.

En el loop infinito `for(;;)`, se evalúa cada *nibble* y se actualizan en la variable **portVble**, para luego actualizar el valor del puerto.

La variable **pBck** se usa para actualizar temporalmente los bits que deben ser borrados y activados, antes de actualizar el valor del puerto del microcontrolador, de esta forma el Puerto no tendrá valores inválidos de forma temporal, sino que pasará de un valor de *nibble* a otro

valor válido.

Como puede observarse el número de bits a rotar puede ser una constante o también una variable como en este caso lo hace la variable **rotaBit**.

6 Nibble: 4 bits.

5.9 LOS OPERADORES DE APUNTADOR & y *

Los operadores & (dirección de la variable...) y * (contenido de la dirección ...) son útiles para obtener la dirección a la cual le corresponde una variable o accesar el contenido de un apuntador o dirección.

En general un **apuntador** es una localidad en memoria que contiene la dirección donde está almacenado un dato.

Todos los apuntadores tienen la misma dimensión en memoria, en este caso 2 bytes, uno conteniendo la parte alta de la dirección y otro la parte baja.

Los apuntadores son ampliamente usados en C, debido a que dan mayor eficiencia al manejo de datos, permiten el manejo de grupos de datos como las estructuras de forma referenciada, sin necesidad de mover todos los datos a procesar en una función.

El uso de apuntadores es un poco más complejo que el manejo convencional por valor, debido a que el apuntador no es el dato, sino la dirección del dato; sin embargo, con disciplina y un poco de concentración se puede obtener una rutina clara y sencilla.

Una variable de tipo *apuntador* es una variable específicamente declarada para contener una dirección hacia un valor de **tipo específico**. Conocer la dirección de una variable puede ser de gran ayuda en ciertos tipos de rutinas. Los *apuntadores* son la base del trabajo con uniones y estructuras de datos.

El tema de los apuntadores será tratado más adelante, sin embargo, acá se presentan estos dos operadores que son usados para manipularlos.

El operador & es un operador unitario (esa es su diferencia con el booleano AND), este se antepone a una variable y entrega la dirección de esa variable (*ver Gráfico 5.1*).



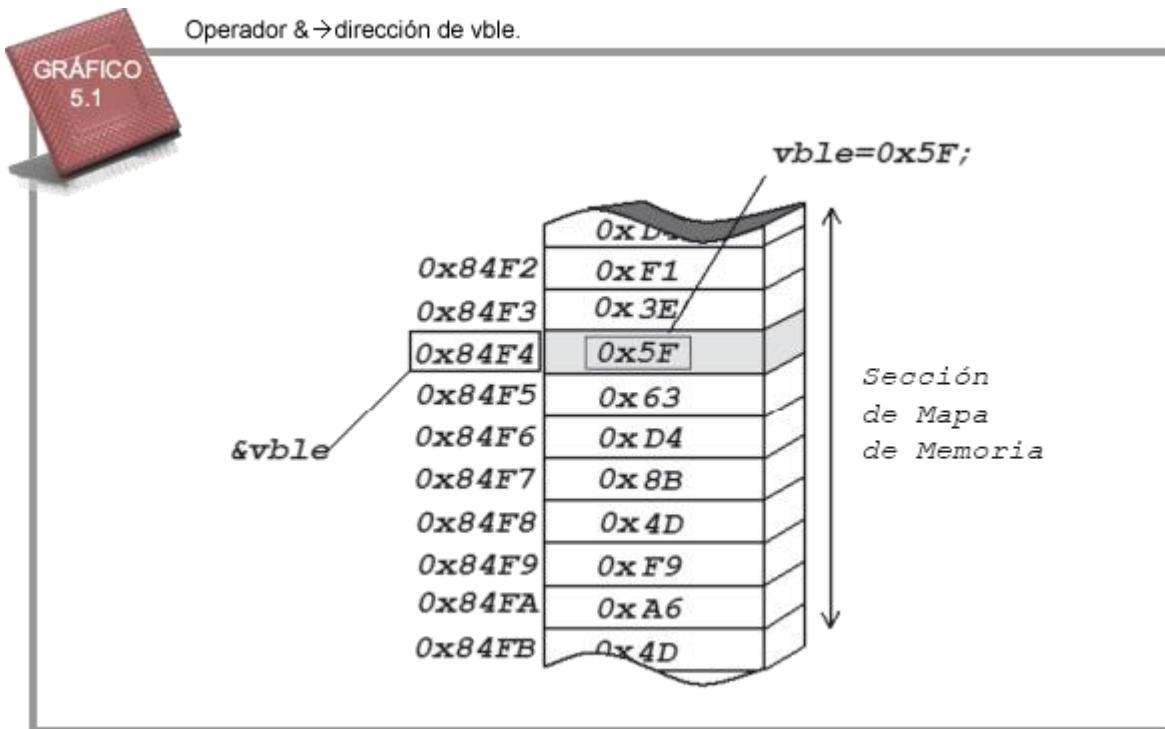
Conocer la dirección de una variable puede ser de gran ayuda en ciertos tipos de rutinas; los apuntadores son la base del trabajo con uniones y estructuras de datos.

El operador & es un operador unitario que se antepone a una variable y entrega la dirección de esa variable.



addr = &vble;

En “addr” quedará la dirección de la variable “vble”.

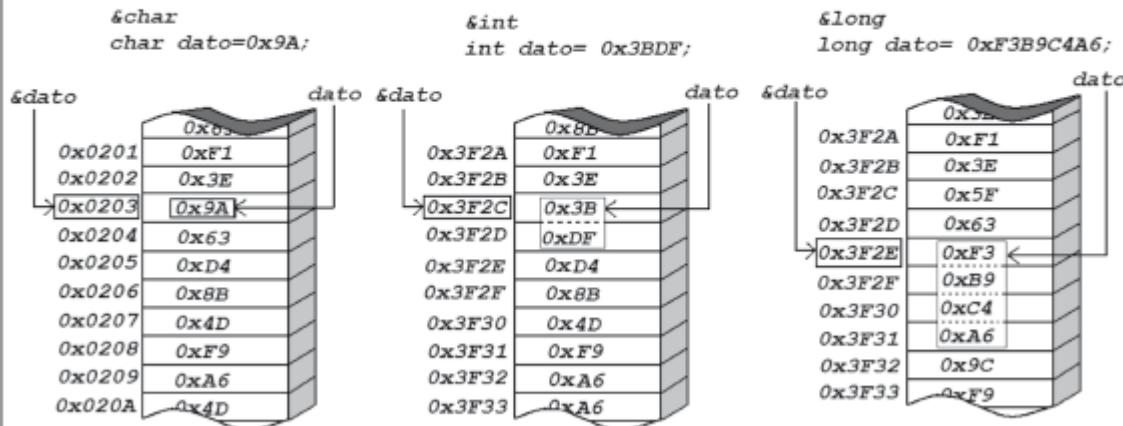


Recuerde que sin importar el tipo de “vble”, su dirección siempre será un valor entero “int”, y por esto la declaración de “addr” deberá ser de tipo apuntador y ocupará 2 bytes.

En el caso de procesadores de 8 bits, el operador & entregará la dirección del byte de mayor peso de la variable (*ver Gráfico 5.2*).



Operador & sobre tipos básicos char, int y long.



Receta

No confundir:
El operador &: como operador booleano es AND, como operador unitario es "la dirección de..." usada para obtener apuntadores.
El operador *: como operador aritmético es multiplicación y como operador unitario es "contenido de la dirección...".

El operador * (contenido de la dirección...) es el complementario de &, es un operador unitario que retorna el valor que contiene la dirección que le sigue.

dato = *addr;

Donde "dato" quedará con el contenido de la dirección "addr".

5.9.1 Declaración de variables tipo apuntador

Las variables que vayan a almacenar direcciones o punteros, se declaran anteponiendo un asterisco * delante del nombre de la variable, esto le indicará al compilador que va a contener un apuntador.

El tipo y modificador deberá ser el tipo de datos al cual apuntará.

modifs tipo *nombrePtr;

En donde:

modifs: modificadores de tipo de dato.

tipo: tipo de dato a ser apuntado.

nombrePtr: es el nombre del apuntador.

Es recomendable que al declarar una variable tipo apuntador, la variable tenga la terminación “Ptr” con el ánimo de poder identificarlo como tal dentro del código.

Así por ejemplo, si se quiere declarar un apuntador a un dato tipo **char**, la sintaxis será:

```
char *apuntadorPtr;
```

En este caso **apuntadorPtr** no será un carácter, sino un entero que va a contener la dirección de un carácter **char**.

La declaración de apuntadores puede mezclarse con las declaraciones normales de datos así:

```
unsigned int vble1, *vble2Ptr, vble3;
```

En este caso “vble1” y “vble3”, quedarán declarados como un tipo **unsigned int**, mientras que “vble2Ptr” quedará declarada como un entero que es una dirección cuyo valor apuntado es un **unsigned int** (*ver Gráfico 5.3*).

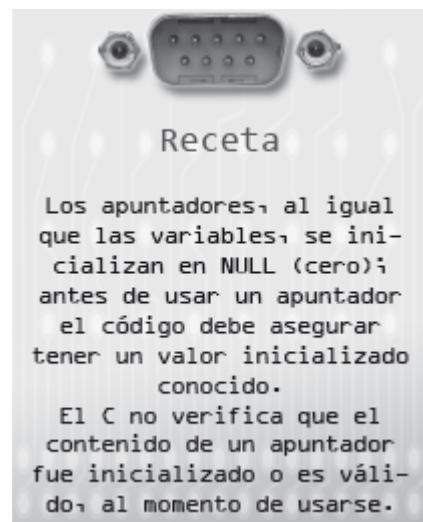
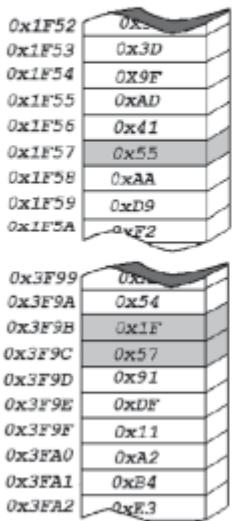


GRÁFICO 5.3

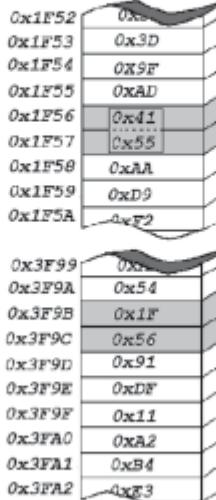
Apuntadores a tipo de datos básicos en C.

```
char *dato1Ptr;
dato1Ptr = 0x1F57;
```

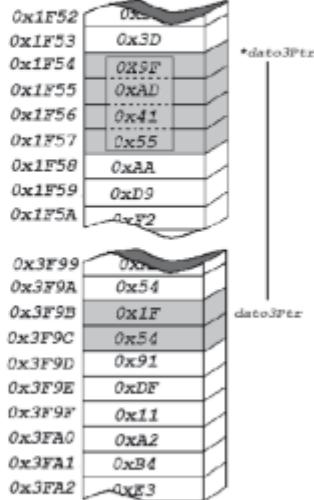


*dato1Ptr --> 0x55

```
int *dato2Ptr= 0x1F56; long *dato3Ptr= 0x1F54;
```



*dato2Ptr --> 0x4155



*dato3Ptr --> 0x9FAD4155

5.10 PROCEDENCIA Y ASOCIATIVIDAD

La precedencia y asociatividad se refiere a la forma como el compilador, y por consiguiente la máquina final, ejecutará las operaciones en una misma expresión.

Así por ejemplo, si se considera la siguiente línea de código:

a = b*2+4-c/8+d++;

Pueden existir múltiples interpretaciones sobre la operación que hace el procesador primero, de tal forma que el ANSI C especifica las reglas de ambas; de la precedencia y de la asociatividad de todos los operadores del C. A continuación en la tabla 5.10 se presenta un resumen de dichas reglas: el operador superior en la tabla tiene la mayor precedencia, los operadores que están en la misma línea tienen la misma precedencia, sin embargo, son examinados de izquierda a derecha.

El operador () se refiere a los paréntesis que cierran los

La tabla de precedencias de los operadores indica que aquellos ubicados en la parte superior tienen la mayor precedencia y los operadores que están en la misma línea tienen la misma precedencia, pero son evaluadas de izquierda a derecha.

argumentos de una función. El operador [] se refiere a los corchetes que encierran el argumento de un array.

TABLA 5.10	Precedencia de operadores en C.
() [] -> .	
! ~ ++ --- + - * & (type) sizeof	
* / %	
+ -	
<< >>	
< <= > =>	
== !=	
&	
^	
&&	
? :	
= += -= *= /= %= &= ^= = <<= >>=	
,	

Nótese que la más alta de las precedencias la tiene el paréntesis () y el corchete [].

De tal manera que le permite al programador forzar las operaciones que desea que se salgan de la forma que especifica la tabla de precedencia (Tabla 5.10).

La segunda de las prioridades la ocupa los operadores unitarios, !,++ ...

Como se puede apreciar la precedencia requeriría ser memorizada para poder ser usada e interpretada, lo cual podría dar lugar a interpretaciones de las operaciones de forma equivocada, a un difícil mantenimiento y difícil de compartir.



Receta

Toda expresión con operadores deberá llevar los () paréntesis en cada operación, esto evita recordar la precedencia y facilita el mantenimiento de los programas desarrollados en C.

Se recomienda en este caso acompañar por () paréntesis, **TODAS** las operaciones en las que se tengan varias variables, expresiones y constantes. De esta forma la precedencia la definirá el programador de acuerdo a lo que requiera en su evaluación.

Así que el ejemplo original se puede obligar a que sea de la siguiente manera:

`a = (b*2) + ((4-c) /8) +(d++);`

O bien se puede obligar a que la expresión sea evaluada así:

`a = b* (2+ (4 - (c / (8+(d++))));`

RESUMEN DEL CAPÍTULO

Las directivas de compilación permiten dar claridad al software desarrollado, estas instrucciones se desarrollan y resuelven en tiempo de compilación, por lo que agregarlas no atenta contra el desempeño del procesador, pueden dar mayor posibilidad de mantenimiento al código y lo independizan del hardware usado.

La directiva #include permite compartir información dentro de los archivos de cabecera, #define permite el manejo de igualdades EQU y declaración de macros.

Con las directivas de compilación condicional como #if, #else, #ifelse, #ifdef y #ifndef, es posible, sobre un mismo archivo, variar el contenido a compilar, lo cual permite administrar la convivencia de versiones de fácil depuración y versiones finales o de producción de un sistema embebido.

Por su parte las directivas #pragma, administran la localización de programa y datos en diferentes secciones del mapa de memoria para optimizar al máximo la memoria limitada del microcontrolador.

El operador en tiempo de compilación *sizeof* entrega el número de bytes ocupados por una expresión. La definición de tipos especiales realizada con *typedef* (definición de tipo), útil incorporarlo en líneas de código que puedan cambiar de longitud, variables que puedan cambiar en un futuro y para facilitar la parametrización de un código.

C soporta los operadores aritméticos estándar para suma +, resta -, multiplicación *, división entera / y residuo % , operadores relacionales mayor >, menor <, igual == diferente !=, y los operadores booleanos OR ||, AND & y NOT !.

El uso de los operadores es flexible y puede relacionar variables a nivel de expresión y de bits, sin embargo, deberá tenerse especial atención sobre el tipo de operación que se requiere realizar en un procedimiento debido a que algunas expresiones, aunque puedan estar sintácticamente bien, pueden no ser la intención del programador.

Los operadores & y * como unitarios entregan la dirección y el contenido de un apuntador, que corresponde a una dirección en la cual se encuentra un dato, un concepto indispensable cuando se manipulan variables de diferente tipo y tablas de datos. Su declaración soporta los mismos tipos y modificadores de las variables.

Existe en C reglas de asociatividad y precedencia que resuelven de forma única una expresión que contiene varios operadores; sin embargo, las reglas se pueden obviar cuando la asociatividad se maneja de forma manual con la pareja () que es el de mayor prioridad. No solo útil para el manejo de la asociatividad sino además, para dar mayor legibilidad al código una vez que algún integrante del equipo programador tome el código para realizar modificaciones.

PREGUNTAS DEL CAPÍTULO

Enumere una ventaja y una desventaja de usar un macro utilizando #define sobre la generación de una subrutina que realice el mismo procedimiento. ¿En qué situaciones usar una o la otra?

¿En qué situaciones de la programación en equipo puede ser útil incluir la directiva #error?

¿Por qué razón no será necesario memorizar la precedencia de operadores en C?

Realice un diagrama similar al del Gráfico 5.3 que acceda el dato **dato = **datoPtr**, donde **datoPtr** es un apuntador a un apuntador a dato entero. Realice un ejemplo que ilustre su uso.

Existe un caso en que el operador desplazamiento ingresa “1”S en lugar de “0”S, ¿por qué el C lo realiza de esta forma?

INTRODUCCIÓN

El capítulo muestra la estructura de los procedimientos en los que está dividido cualquier proyecto: las funciones, su declaración y llamado dentro de la estructura del programa.

Se discute de forma individual cada una de las sentencias de control que constituye el lenguaje C, aplicando cada una al mundo embebido, con ejemplos exactos que muestran su uso y cuidados especiales.

Muchas de ellas manejan secciones basadas en bucles, como son el for, while y el do{ }while; otras se basan en condiciones, como son el if{ }else{ } y el switch; algunas otras, no muy recomendadas, afectan el flujo normal del programa como lo son break, continue, goto y return.

Se trata además el tema que permite la coexistencia entre el lenguaje ensamblador y el código en C, las razones que llevan a usar un lenguaje de ensamblador y se muestran algunas técnicas para hacerlo de forma segura.

Luego se cubre el tema de la recursividad, su ventaja y también su limitación, para llegar a los arreglos de datos numéricos (*arrays*) y de caracteres ASCII (*strings*).

En el aparte 6.7 se retoma el tema de las interrupciones, ya no como concepto sino de la forma como se implementan en un ejemplo real, algunas consideraciones importantes y sintaxis propia en el Codewarrior^R.

Se culmina el capítulo con las convenciones útiles de programación en C, las cuales no hacen parte de la sintaxis, pero sí de normas que se recomiendan, en especial cuando se está trabajando en equipo.

6.1 FUNCIONES EN C PARA SISTEMAS EMBEBIDOS

La función es el corazón de un programa en C, corresponde a lo que en lenguaje ensamblador se llama un procedimiento o una subrutina invocado mediante las instrucciones **BSR** (*Branch To Subroutine*) para llamados cortos, o **JSR** (*Jump To Subroutine*) o **CALL** para llamados a rutinas en direcciones lejanas.

El objetivo de una función es proveer un mecanismo que permita invocar una secuencia de código, el cual puede ser llamado varias veces en un programa, facilitando el entendimiento del código y creando además código reusable, evitando el tener que incluir dicho código cada vez que requiere su ejecución

La forma general de una función es:

```
retorno Nombre_Funcion(argumento1, argumento2 „„ , argumentoN){
```

Cuerpo de la Función



Con una función el programador que trabaja en lenguaje C, puede hacer un llamado a una secuencia de código, la cual puede reutilizar nuevamente cada vez que requiera que dicho código se ejecute.



}

6.1.1 Prototipo de una función

El prototipo de una función se define como la **forma** de la función, se refiere solo a la línea que contiene su retorno, el nombre de la función y sus argumentos, terminando con un punto y coma:

retorno Nombre_Funcion(argumento1, argumento2, ... , argumentoN);

En donde:

retorno: es el valor que retorna la función después de invocada. Si la función no retorna valor alguno, deberá tener la palabra reservada **void**.



Para construir una función el programador le asigna un nombre, un valor de retorno cuando la función es invocada y una serie de argumentos sobre los cuales la función ejercerá alguna operación.

Aunque teóricamente una función puede recibir cualquier número de argumentos, para los sistemas embebidos de 8 bits no se recomienda incluir más de cuatro, porque cada argumento genera código adicional que hará más lenta su ejecución.



Nombre_Funcion: es el nombre de la función, la define el programador.

argumento1, argumento2... argumentoN: son las variables separadas por coma, que se entregan a la función y sobre las cuales se realizará alguna operación. En caso de no requerir argumentos, la función deberá tener en los argumentos: (**void**).

Una función puede, en teoría, recibir cualquier número de argumentos, los cuales se entregan normalmente a la función en los registros de trabajo del microcontrolador, y en caso de ser estos insuficientes, se entregan en el *stack*.

En la práctica y específicamente para sistemas embebidos de 8 bits, no se recomiendan funciones que reciban más de 4 argumentos, debido a que cada argumento generará código adicional antes de hacer el llamado a la función.

El **retorno** se refiere al resultado que entrega la función al final de su ejecución, también se le denomina el resultado de la función, el cual se entrega mediante la palabra reservada **return** seguido de su valor, este solo puede ser **uno y solo uno** de los tipos de variables estándar: **char, int, long, float**, acompañado de uno, a más de los modificadores estándar de las variables, o igualmente pueden retornar un apuntador (concepto que se verá más adelante).

En el caso que una función no necesite retornar un valor, su prototipo debe indicar explícitamente la palabra **void** en su retorno, para indicar que no cuenta con retorno, de modo que al hacerse el llamado externo no se espera ningún valor de regreso no requiriéndose de la palabra **return** en el cuerpo de la función.

Si un prototipo de función se declara sin retorno o sin argumento, el C asumirá que tanto el retorno como su argumento son de tipo **int (valor entero de 16 bits)**. Aunque esta situación no genera un error en el compilador, si puede generar un Warning que podría llegar a ocasionar confusiones o datos erróneos de entrega o retorno a las funciones que hacen llamado a estas funciones.

Funcion();

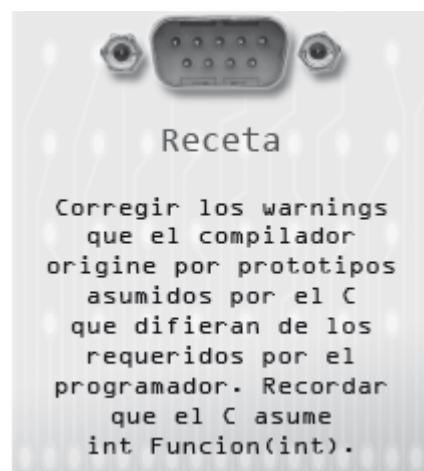
Este prototipo es asumido por el ANSI C como:

int Funcion(int valor);

Si una función no retorna y no recibe valor alguno deberá declararse como:

void Funcion(void);

El prototipo de una función debe siempre aparecer en la parte superior a su invocación o llamado, de lo contrario el prototipo no será encontrado, o será asumido como **int Funcion(int arg1);** lo cual puede al final generar un error o (peor aún), un *Warning* en el compilador, en el cual los prototipos asumidos y encontrados al momento del *linker* no coincidirán.



```
char vble2; // declaración de una variable global

void Funcion1(void){

char vble1;

vble1 = 0;

Funcion2();           // llamado a Funcion2

                    // NO encuentra su prototipo en la parte superior

                    // de tal forma que lo asume int Funcion2(int arg1);

}

void Funcion2(void);

void Funcion2(void){

vble2 = 4;

}
```

Siendo la forma correcta:

```
char vble2;

void Funcion2(void);

void Funcion1(void){

char vble1;

vble1 = 0;

Funcion2();

}

void Funcion2(void){

vble2 = 4;

}
```

Cuando una función retorna un valor, este deberá entregarse mediante la palabra reservada **return**, seguido del valor a retornar, que deberá ser del mismo tipo del que retorna la función o tener el correspondiente molde de ajuste (*cast*).

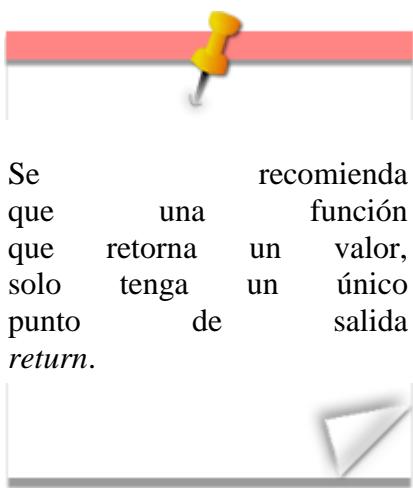
Así la función:

```
unsigned int Funcion3(void){  
    unsigned int vInt;  
    return vInt;  
}
```

La función **Funcion3()**, retorna el valor de la variable **vInt**, que es del mismo tipo declarado de la variable.

Si en medio del código de una función aparece su retorno, este abortará el código que sigue antes de terminar la función y realiza el retorno **RTS** (*Return From Subroutine*) de inmediato, o **RTI** (*Return From Interrupt*), en caso que se trate de una función de atención a una interrupción.

Una función puede tener varios puntos de salida **return**, sin embargo, por cuestiones de programación estructurada, se recomienda que solo tenga un punto de salida **return**, excepto si se trata de alguna optimización de velocidad o de uso de memoria.



Los prototipos de una función pueden aparecer en uno o varios archivos las veces que sean necesarios, y no generarán ningún código o afectarán el tiempo de ejecución de la máquina, pero por cuestiones de orden, solo deberían aparecer en los archivos de cabecera (.h) o en la función que los usa en caso de ser estáticos (*static*).

Si una función va a recibir argumentos, estos son agregados por el código del compilador antes de hacer el llamado a la función; el lugar en que se entregan los argumentos es el mismo en el cual el cuerpo de la función espera estos valores, algunos se entregarán en los registros internos de trabajo, sea en el **AccA**, en el Registro H o registro X o en ambos en el caso de un entero, o en el stack en un orden preestablecido.

6.1.2 El concepto de paso de argumentos a función

Pasar argumentos a una función consiste en entregarle los valores con los que va a realizar la operación.

Para ilustrar este concepto de paso de argumentos considérese este ejemplo de la vida real:

EJEMPLO

Imagínese una gran biblioteca, con muchos libros, de diferentes temas, editoriales, autores y fechas de publicación.

En la biblioteca está el bibliotecario y un empleado o funcionario, que se encarga de organizar los libros.

*El bibliotecario tiene varias funciones, una de ellas es encargar la función de organizar los libros a su empleado de acuerdo a varias indicaciones, y el empleado al terminar la labor, deberá responder por el resultado de su función, con una respuesta de cómo quedó dicha labor. El bibliotecario requiere inicialmente tomar 4 libros y entregarlos al empleado para que este los organice en forma alfabética, antes de llamar al empleado, debe tomar los libros en sus manos, llamarlo y **pasarle los libros de forma física** para que sean organizados. Esta labor es sencilla y no toma mucho tiempo, y el bibliotecario lo puede hacer con facilidad, inclusive si son 5 ó 6 libros, el bibliotecario podría tomarlos en sus manos y entregarlos al empleado para que los organice; sin embargo, cuando se trata de organizar un armario completo de 200 libros, no resulta práctico para el bibliotecario transportar todos los libros a organizar, esto le tomaría demasiado tiempo al bibliotecario y tal vez pueda tomar un tiempo mayor o similar a la labor del empleado, además que no podría con sus dos manos tomar los 200 libros para entregarlos al empleado; en este caso resulta mejor **señalar la ubicación del primer libro** a organizar y llamar al empleado y pedirle que organice los libros en orden alfabético empezando por el primer libro que fue señalado y hasta que encuentre el último libro dentro de ese armario.*



Pasar los argumentos por valor significa que el programa principal toma los datos uno a uno y los pasa a la Funcion() respectiva.



En el ejemplo anterior el bibliotecario hace las labores del programa principal main(), el empleado es la función Funcion(), que es llamada por el programa principal.

Pasar los argumentos por referencia indica que el programa entrega una ubicación única a la Funcion() respectiva a partir de la cual se encuentran los datos.



En la primera parte donde el bibliotecario **pasa los libros de forma física**, se dirá que está **pasando los argumentos por valor**, mientras que en el segundo caso, donde es solo **señalada la posición del primer libro** a organizar, se dirá que se **pasa el argumento por referencia** o por dirección.

6.1.3 Paso de argumentos por valor

Este método copia el valor del argumento en la posición esperada por la función, de tal forma que si se esperan N argumentos, estos N valores serán copiados o movidos a las localidades esperadas por la función, los valores originales no cambian su valor antes o después del llamado a la función.

Considérese el siguiente ejemplo:

EJEMPLO No. 13

Paso de argumentos por valor

Objetivo:

Escribir una rutina en C que retorne el resultado de la suma de 3 argumentos tipo **unsigned char**, invocar la función pasando los valores de cada argumento.

Solución:

```
***** Ejemplo 13 *****
// Paso de argumentos por valor
// Fecha: Feb 4,2009
// Asunto: Suma de valores.
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// Freescale™ o PIC-Link(2008-12-15) Microchip™.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#define __FREESCALE_
##define __PIC_
#ifndef __FREESCALE_
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP
#endif
#ifndef __PIC_
#include "argsValor.h"
typedef int32 ulong;
typedef unsigned char uchar;
#endif
volatile ulong resultado;
unsigned long FxSuma(uchar arg1,uchar arg2,uchar arg3){
    return (ulong)arg1+(ulong)arg2+(ulong)arg3;
}
void Mcu_Init(void){
#ifndef __FREESCALE_
    Disable_COP();
#endif
#ifndef __PIC_
    Mcu_Init_PIC();

```

```

#endif
}
void main(void) { //Función principal
    Mcu_Init();
    resultado = FxSuma(2,21,45);
    for(;;) {
        /* loop forever */
    }
}

```

Discusión:

El código generado en ensamblador del HCS08 será el siguiente:

```

47:     resultado = FxSuma(2,21,45);
0002 450000 [3] LDHX @resultado
0005 89 [2] PSHX
0006 8b [2] PSHH
0007 a602 [2] LDA #2
0009 87 [2] PSHA
000a a615 [2] LDA #21
000c ae2d [2] IDX #45
000e ad00 [4] BSR FxSuma
0010 a703 [2] AIS #3
0012 450000 [3] LDHX @resultado
0015 89 [2]
0016 8b [2] PSHH
0017 a604 [2] LDA #4
0019 cd0000 [5] JSR _COPY
001c L1C
48:     for(;;{
001c 20fe [3] BRA L1C ;abs = 001c
49: } /* loop forever */
50: }

```

El Argumento 2 en el stack

El Argumento 21 en AccA

El Argumento 45 en X

El retorno se almacena en el long resultado

Se puede observar como las constantes 2, 21 y 45 son copiadas en los registros y posiciones de memoria donde el cuerpo de la función espera dichos valores para ser operados, en la medida en que se pasen más argumentos, el código previo al llamado de la función **FxSuma** es mayor, consumiendo más memoria de programa y aumentando el tiempo de ejecución.

El resultado o retorno, resulta del llamado de la función **FxSuma**.

Para el caso de Microchip™, el código generado será:

```

235 ..... resultado = FxSuma(2,21,45);
236 008E: MOVLW 02
237 008F: MOVWF 25
238 0090: MOVLW 15
239 0091: MOVWF 26
240 0092: MOVLW 2D
241 0093: MOVWF 27
242 0094: GOTO 040
243 0095: CLRF 23
244 0096: CLRF 22
245 0097: MOVF 79,W
246 0098: MOVWF 21
247 0099: MOVF 78,W
248 009A: MOVWF 20
249 ..... for(;;)
250 ..... /* loop forever */
251 009B: GOTO 09B
252 .....
253 .....
254 009C: GOTO 09C

```

En general, una función puede recibir tantos argumentos como los declare su prototipo, sin embargo, se debe recordar que la adición de un nuevo argumento a una función genera más código extra previo al llamado de la función. Por esta razón no resulta práctica una función que reciba más de 4 argumentos, en este caso se debe buscar la opción de hacerlo por referencia como se verá a continuación.

6.1.4 Archivos de cabecera (.H)

Todo proyecto, por sencillo que parezca, deberá estar conformado por varios módulos, a fin de darle orden al proyecto completo. La división por módulos permite organizar equipos de trabajo, cambiar un módulo de forma sencilla e identificar problemas en depuración en el módulo que causa el funcionamiento del programa por fuera de las especificaciones requeridas.

Así por ejemplo, en un equipo celular se tiene un módulo para el manejo del **display**, otro para el **teclado**, uno que se encarga de la batería, otro del módulo **bluetooth**, de su puerto **infrarrojo, USB**, de la información **estadística** y de contactos, manejo de la **cámara** de imágenes y video. Cada uno de los módulos anteriores tiene asociado un archivo con extensión .C y otro con extensión .H que corresponde a su archivo de cabecera.

El archivo con extensión .H puede contener varios elementos:

Los prototipos de las funciones públicas que otros módulos usarán.

Las variables globales que serán compartidas a otros módulos.

Los macros que dependen del hardware y que son referenciados por el módulo mismo o por otros módulos.

Todo archivo de extensión .C que requiere referenciar las declaraciones de otro módulo, puede incluirlas usando la directiva **#include** seguida por el nombre del archivo de cabecera del módulo requerido .H.

Si el archivo de cabecera se encuentra en el mismo directorio del archivo que referencia entonces el nombre del archivo de cabecera se debe encerrar entre comillas dobles “ ”.

```
#include "modulo.h"
```

Si por el contrario, el archivo se encuentra en el directorio de las librerías definido por el proyecto, su inclusión se hará encerrando el nombre del archivo de cabecera por los signos < > así:

```
#include <modulo.h>
```

La declaración anterior va ubicada al inicio del archivo fuente .C.

Debido a que un proyecto puede tener varios archivos de cabecera, y crecen de acuerdo a su complejidad, en algunos casos se genera un archivo general de cabecera .H, que contiene a su vez todos los archivos de cabecera necesarios para el proyecto, de esta forma los archivos fuente, .C, solo requieren incluir este único archivo. El archivo de cabecera general puede lucir así:



Cada vez que una función involucra un argumento se genera un código de llamada para ese argumento. Si son veinte argumentos, serán veinte códigos previos a la llamada, lo que implica un gasto innecesario de memoria; en vez de ello, es mejor hacer un llamado por referencia.

El archivo de cabecera (.H) contiene las variables, las funciones y los macros globales que serán utilizados por otros módulos.



```
// celular.H

#include <stdio.h> // librería de I/O de C

#include <stdlib.h> //librería estándar de C

#include "display.h"

#include "teclado.h"

#include "bateria.h"

#include "bluetooth.h"

#include "infrarrojo.h"

#include "usb.h"

#include "estadistica.h"

#include "camara.h"
```

6.1.5 Paso de argumentos por referencia



En el paso de argumentos por referencia la función recibe la dirección de dónde encontrar los valores a operar, no los valores mismos.



La llamada por referencia sucede cuando no se pasan uno a uno los valores a operar en la función, sino que en su lugar se pasa la dirección en la cual están los valores a operar.

Esta forma de pasar los argumentos es mucho más general y rápida, sobre todo cuando los argumentos son varios. Se trata de una estructura o de una cadena de caracteres la cual puede ser de cantidad variable.

Una dirección siempre es de 2 bytes (entero), con lo que no importa el número de datos a operar, siempre su tiempo de ejecución y preámbulo antes de hacer el llamado a la función será igual.

De la misma forma que el paso de argumentos por valor, el paso por referencia puede recibir un número indefinido de referencias (o de apuntadores); sin embargo, en la práctica no se recomienda pasar más de 4 referencias o argumentos, debido a que no lo hace práctico ni eficiente.



EJEMPLO No. 14

Paso de argumentos por referencia

Objetivo:

Escribir una rutina en C **FxOperacion**, la cual recibe 4 argumentos: el tipo de operación a realizar entre los argumentos 2 y 3 siguientes de tipo **unsigned long**, y posicione el resultado en el argumento 4. Todos los argumentos, excepto el tipo de operación, deberán entregarse por referencia. Si el resultado de alguna de las operaciones es cero, la aplicación deberá encender el Led de la salida OUT-1.

Solución:

```
***** Ejemplo 14 *****

// Paso de argumentos por referencia

// Fecha: Feb 4,2009

// Asunto: Operaciones por referencia.

// Hardware: Sistema de desarrollo AP-Link para

// FreescaleTM y PIC-Link para MicrochipTM.

// Versión: 1.0 Por: Gustavo A. Galeano A.

*****



#define __FREESCALE_

#define __PIC__

#ifndef __FREESCALE__

#include <hidef.h> /* for EnableInterrupts macro */

#include "derivative.h" /* include peripheral declarations */

#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP

#define Led1On() PTC_PTC3 = 1; DDRC_DDRC3 = 1
```

```

#define Led1Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1

#endif

#ifndef __PIC__

#include "argsReferencia.h"

#define Led1On() output_high(PIN_B4)

#define Led1Off() output_low(PIN_B4)

typedef int32 ulong;

#endif

#define SUMA 0

#define RESTA 1

#define MULT 2

#define DIV 3

#define MOD 4

void FxOperacion(char operac, ulong *arg1Ptr, ulong *arg2Ptr,
                  ulong *resultPtr){

    if(operac == SUMA){ *resultPtr = (*arg1Ptr) + (*arg2Ptr); }

    if(operac == RESTA){ *resultPtr = (*arg1Ptr) - (*arg2Ptr); }

    if(operac == MULT){ *resultPtr = (*arg1Ptr) * (*arg2Ptr); }

    if(operac == DIV){ *resultPtr = (*arg1Ptr)/(*arg2Ptr); }

    if(operac == MOD){ *resultPtr = *arg1Ptr%*arg2Ptr; }

    if(*resultPtr == 0){ Led1On(); }else{ Led1Off(); }

}

ulong vble1,vble2,vble3;

void main(void) { //Función principal

```

```

#define __FREESCALE__
    Disable_COP();
#endif
#define __PIC__
void Mcu_Init_PIC(); /*el cuerpo está en el Ejemplo 4*/
#endif
for(;;) {
    vble1 = 1520;
    vble2 = 321;
    FxOperacion(SUMA,&vble1,&vble2,&vble3);
    vble1 = 0;
    vble2 = 0xFFFF;
    FxOperacion(MULT,&vble1,&vble2,&vble3);
    vble1 = 50;
    vble2 = 2;
    FxOperacion(DIV,&vble1,&vble2,&vble3);
    FxOperacion(MOD,&vble1,&vble2,&vble3);
} /* loop forever */
}

```

Discusión:

El código previo generado para el HC08 de la función

FxOperacion(SUMA,&vble1,&vble2,&vble3);

Es el siguiente:

```

FxOperacion(SUMA,&vble1,&vble2,&vble3);
0022 87 [2] PSHA Argumento SUMA (por valor)
0023 450000 [3] LDHX @vble1
0026 89 [2] PSHX La dirección de vble1 al stack
0027 8b [2] PSHH
0028 450000 [3] LDHX @vble2
002b 89 [2] PSHX La dirección de vble2 al stack
002c 8b [2] PSHH
002d ae00 [2] LDX @vble3:MSB
002f a600 [2] LDA @vble3 La dirección de vble3 en X:A
0031 cd0000 [5] JSR FxOperacion
0034 a705 [2] AIS #5 Recupera el SP(stack pointer)

```

El modificador volatile se hace necesario para que el Compilador no realice optimización sobre el llamado a la función FxOperacion(), sin este el código resultante solo realiza el llamado a la ultima FxOperacion(MOD,...) por considerar que es la única que tiene efecto sobre la variable vble3.

Este código será el mismo independiente de la longitud en bytes de las variables vble1,vble2 y vble3,porque siempre se pasarán sus direcciones y no sus contenidos, en este caso se están almacenando en el stack 5 bytes y 2 bytes adicionales en AccA y IX, para un total de 7 bytes que se mueven.

Si esta función se invocara por valor, el código previo al llamado de la función FxOperacion sería más largo, debido a que tendría que mover 13 bytes: 11 al stack y 2 a los registros: 3 variables de tipo ulong → $3 \times 4 = 12$ bytes, más 1 byte adicional del argumento char operac.

El código previo generado para el PIC 16F de la función

FxOperacion(SUMA,&vble1,&vble2,&vble3);

Es el siguiente:

```

669 FxOperacion(SUMA, &vble1, &vble2, &vble3);
670 02A7: CLRF 2D   La dirección de vble1 (0x0020) va
671 02A8: CLRF 2F   a (0x2D y 0x2E)
672 02A9: MOVLW 20
673 02AA: MOVWF 2E
674 02AB: CLRF 31
675 02AC: MOVLW 24
676 02AD: MOVWF 30
677 02AE: CLRF 33
678 02AF: MOVLW 28
679 02B0: MOVWF 32
680 02B1: CALL 072  Realiza llamado a FxOperacion

```



Materiales adicionales en la

Código fuente, paso de argumentos por referencia, para **Freescale™** y **Microchip™**.

Los argumentos vble1, vble2 y vble3 que son de tipo unsigned long de 4 bytes se posiciona en 6 bytes, 2 por cada variable que corresponden a su dirección, de esta forma el movimiento de datos previo al llamado de la función FxOperacion es menor.

NOTA: No siempre el llamado por referencia implica menor preámbulo para el llamado de funciones, en este ejemplo si los argumentos son por ejemplo de tipo char (1 byte), resulta más eficiente el llamado por valor el cual moverá 3 bytes por valor, en lugar de 6 bytes por referencia.

6.2 SENTENCIAS DE CONTROL



Las sentencias de control son bloques de programa que definen una funcionalidad, pueden variar su funcionamiento dependiendo del resultado de una operación o el valor de una variable.



En esta sección se explicará una a una las sentencias de control de programa que proporciona el ANSI C, para aclarar cada una de ellas y poner atención sobre algunas excepciones y recetas de este capítulo; además se mostrarán algunos ejemplos prácticos que permitirán al lector un mayor adiestramiento en cada una de ellas.

6.2.1 La sentencia “if .. else”

La forma general de esta sentencia es:

A

```
if(<condicion>){
```

B

Bloque de programa
<condicion>
VERDADERA

```
}else{
```

C

Bloque de programa
<condicion> VERDADERA

```
}
```

D

Una vez que el control del programa llega a la sentencia if, **punto A**, evalúa la condición para examinar si es verdadera (diferente de cero) o es falsa (cero).

Si la evaluación de la condición resulta ser verdadera el control pasará al **punto B**, y ejecutará el bloque de programa VERDADERA, una vez terminado pasa al **punto D** para continuar con la ejecución del programa desde este punto.
Sentencias de control Bloque de Programa



Las sentencias if y else son excluyentes, el programa ejecutará el código asociada a una de ellas, pero nunca las dos.



Si por el contrario, al evaluar la condición resulta ser falsa, el control del programa pasará al **punto C** para ejecutar el bloque de programa FALSA y al terminar al **punto D** de ejecución.

El bloque de programa FALSA es opcional con lo que puede existir un **if sin else** en el cual solo se ejecutará el bloque de programa VERDADERA si la condición es diferente de cero, o no hará absolutamente nada si la condición es falsa.

Recuerde que solo se ejecuta el código asociado al **if** ó al **else**, nunca ambos.

La condición a evaluar puede ser una constante, una variable o una expresión con uno a varios conectores y operandos. Considérese el siguiente ejemplo:

EJEMPLO No. 15

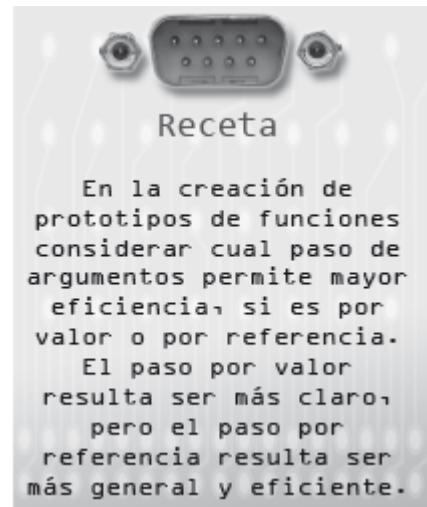
Uso de la sentencia if..else

Objetivo:

Escribir un programa en C que realice operaciones con dos variables tipo **int**, *vble1* y

vble2, dependiendo del estado de las teclas **INPUT-1** y de **INPUT-2**, de la siguiente forma:

1. Si la tecla **INPUT-1** se presiona, el código deberá hacer la suma de *vble1* y *vble2* y



actualizar el resultado de la variable result.

2. Si la tecla **INPUT-2** se presiona, la variable result deberá contener la operación $vble1 - vble2$.

3. Si estando presionada **INPUT-1**, se presiona además **INPUT-2**, el valor de la variable result, deberá ser la multiplicación de $vble1$ y $vble2$.

En todos los casos las salidas **OUT-2** y **OUT-1** deberán indicar el estado de los bits de mayor y menor peso de result, respectivamente.

Si el resultado de la operación **result** en algún caso es cero, deberá indicarlo con la salida auditivaBuzzer **BUZZ**.

Solución:

```
***** Ejemplo 15 *****  
// Uso de la sentencia if...else  
// Fecha: Feb 4,2009  
// Asunto: Operaciones basadas en  
// presiones de la teclas INPUT-1, INPUT-2  
// Hardware: Sistema de desarrollo AP-Link  
// Freescale™ y PIC-Link Microchip™.
```

```

// Versión: 1.0 Por: Gustavo A. Galeano A.

//*****



#define __FREESCALE_

#ifndef __PIC_

#endif __FREESCALE_

#include <hidef.h> /* for EnableInterrupts macro */

#include "derivative.h" /* include peripheral declarations */

#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP

#define Key1_Press() (!PTD_PTD1)

#define Key2_Press() (!PTD_PTD2)

#define Buzzer_On() PTD_PTD0 = 1;DDRD_DDRD0 = 1

#define Buzzer_Off() PTD_PTD0= 0;DDRD_DDRD0 = 1

#define OUT_1_On() PTC_PTC3 = 1;DDRC_DDRC3 = 1

#define OUT_1_Off() PTC_PTC3 = 0;DDRC_DDRC3 = 1

#define OUT_2_On() PTC_PTC2 = 1;DDRC_DDRC2 = 1

#define OUT_2_Off() PTC_PTC2 = 0;DDRC_DDRC2 = 1

#define INPUT_1() Key1_Press()

#define INPUT_2() Key2_Press()

int vble1,vble2;

typedef long long32;

long result;

void Mcu_Init(void){

    Disable_COP();
}

```

```

}

#endif

#ifndef __PIC_

#include "if_else.h"

#define Key1_Press() (!input(PIN_B0))

#define Key2_Press() (!input(PIN_B1))

#define Buzzer_On() output_high(PIN_B2)

#define Buzzer_Off() output_low(PIN_B2)

#define OUT_1_On() output_high(PIN_B4)

#define OUT_1_Off() output_low(PIN_B4)

#define OUT_2_On() output_high(PIN_B5)

#define OUT_2_Off() output_low(PIN_B5)

#define INPUT_1() Key1_Press()

#define INPUT_2() Key2_Press()

typedef int32 long32;

int vble1,vble2; //Definicion de variables

long32 result;

void Mcu_Init(void){ //Inicializacion del MCU

    Mcu_Init_PIC();

}

#endif

void main(void) { //Funcion principal

    Mcu_Init();vble1 = 54;

    vble2 = 23;

```

```

for(;;) {

    if(INPUT_1() && !INPUT_2()){

        result = (long32)vble1+(long32)vble2;

    }

    if(INPUT_2() && !INPUT_1()){

        result = (long32)vble1-(long32)vble2;

    }

    if(INPUT_1() && INPUT_2()){

        result = (long32)vble1*(long32)vble2;

    }

    if(result & 0x01){OUT_1_On();}else{OUT_1_Off();}

    if(result &0x80000000){ OUT_2_On(); }else{OUT_2_Off(); }

    if(result == 0){Buzzer_On();}else{Buzzer_Off();}

    while(!INPUT_1() && !INPUT_2());

} /* loop forever */

}

```

Discusión:

Se inicia con la definición de las variables y unos valores aleatorios, en el ciclo **for** (;;) el código quedará verificando la presión de alguna de las teclas para realizar la operación respectiva. Se puede observar que el código en C corresponde al enunciado usando las secuencia if...else. Observese además como para dar mayor legibilidad al código, se hace una redefinición de INPUT _1, basado en otros #define.

Existe un error muy común en este tipo de sentencias, el cual consiste en hacer la evaluación con una sola igualdad `=`, en lugar de doble igualdad `==`.

Como se recordará el primer igual hace una asignación a la variable a la izquierda de `“=”`, mientras que la doble igualdad `“==”` solo evalúa o examina si el valor de la variable de su izquierda es EXACTAMENTE igual al valor de la derecha.

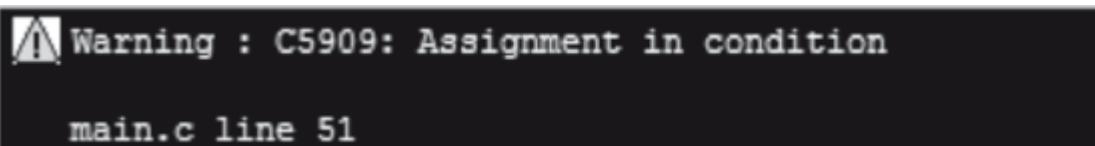
Este aspecto es importante debido a que el compilador NO genera un error de sintaxis, porque ambas expresiones son perfectamente válidas, sin embargo, en ejecución el programa puede estar haciendo algo diferente a lo que el programador deseaba.

Así por ejemplo, si se omite uno de los iguales en la sección de código:

```
if(result = 0){ //asignación  
    Buzzer_On();  
}else{  
    Buzzer_Off();  
}
```

El compilador NO generara error, pero si los siguiente warning:

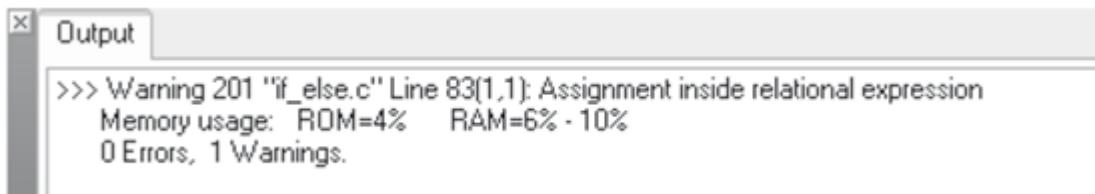
En Codewarrior® para Freescale™:



The screenshot shows a terminal window with a black background and white text. It displays a warning message from the compiler: "Warning : C5909: Assignment in condition". Below this, it specifies the file "main.c" and line number "51".

```
Warning : C5909: Assignment in condition  
main.c line 51
```

En CCS para Microchip™:



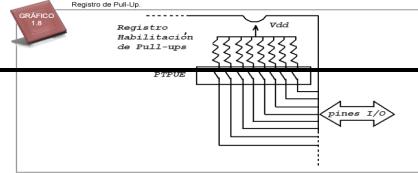
Al llegar el flujo del programa a la condición if, lo primero que hace es resolver la expresión, en la cual realiza la asignación “=” a la variable result, y con esto la variable es borrada, luego interroga ese valor, resultando ser FALSO siempre, y con esto se ejecutará el código else SIEMPRE.

Si el programa es compilado así:

```
if(result == 0){ //evaluación  
    Buzzer_On();  
}  
else{  
    Buzzer_Off();  
}
```

Al llegar a la evaluación del if, el programa NO modificará el valor de la variable result, sino que preguntará si el valor es exactamente igual a 0.

En ambos casos el programa compiló bien, pero ejecutará líneas muy diferentes.



El uso de la sentencia if...else es uno de los más útiles en la programación estructurada de cualquier lenguaje de programación, debido a que permite tomar decisiones basadas en el resultado de una expresión

o valor, sin embargo el uso de más de 3 if anidados hace que el código no sea muy legible debido a que en cada uno de ellos se debe conservar como mínimo un tabulador, y hace que el código se extienda de forma horizontal y dificulta un poco la visibilidad en el editor.

Si la necesidad es usar los if anidados, se recomienda o bien usar una sentencia mas vertical como lo es la switch o bien crear funciones dentro de los if que faciliten la visibilidad.



Es importante considerar que si quiere evaluar y recordar el uso del operador “=” para asignar y el operador “==” es exactamente igual a... recordar las Recetas del Capítulo 5 .

Al utilizar una serie de if o else es importante establecer la jerarquía cuando se presentan de forma anidada. Para ello lo mejor es utilizar los {} para hacer la indicación respectiva.

Existe una forma abreviada de la sentencia if... else, recomendada solo cuando los bloques de programa son cortos y solo son expresiones, nunca otra secuencia de C.

En forma general: <condicion> ? ExpresionTRUE : ExpresionFALSE

La secuencia de los ejemplos puede entonces digitarse así:

```
Pin_Mcu() ? vble1 = 50:vble1++ (vble1 == 4) ? vble2 = vble1+10
: vble2 = 30
```

Una última recomendación sobre el if consiste en usar siempre los corchetes {} de cada uno de ellos cuando se tienen if anidados. De esta forma nunca se tendrán confusiones sobre si un else pertenece a un if externo o interno.

Así por ejemplo, una expresión como:

```
if(vble1)
    if(vble2) vble1 = 0;
    else vble1 = 2;
```

¿A cuál de los dos if, pertenece el único **else**? Por norma del C pertenece al segundo if, sin embargo, con el uso de los corchetes { }, se puede ser más claro aún e indicar exactamente lo que desea el programador y no lo que el C asume; esta claridad es importante porque si en el futuro otros programadores trabajan sobre el código lo podrán entender con facilidad para modificarlo, de tal manera que se pueden tener ambas expresiones:

```
if(vble1){
if(vble2)
    vble1 = 0;
else      vble1 = 2;
}

O bien:
if(vble1){
    if(vble2) vble1 = 0;
}else{
    vble1 = 2;
}
```

6.2.2 La sentencia “do... while”

La secuencia **do... while** suele usarse cuando se quiere dejar el programa ejecutando una secuencia de líneas en un ciclo, mientras no se cumpla una condición específica.

La forma de la sentencia es:

A

do{

B

}while(<condicion>);

Bloque de programa DWH

D



La sentencia do... while es útil cuando se quiere dejar una función en un ciclo repetitivo, y sólo saldrá de dicho ciclo cuando se cumpla la condición establecida por el programador.



Al llegar el control del programa al punto A, pasa de inmediato a ejecutar el **Bloque de Programa DWH**, ejecutándolo completamente. Al terminar evalúa la <condicion>, si esta resulte ser verdadera, volverá al punto B para ejecutar de nuevo el **Bloque de Programa DWH** y así hasta que llegando a la <condicion> esta resulte ser falsa.

Nótese que el **Bloque de Programa DWH** se ejecuta como mínimo una vez, independiente del estado de la <condicion>, de hecho, si al llegar al punto A la condición es falsa, el **Bloque de Programa DWH** se ejecuta una vez.

También téngase en cuenta que la condición solo tendrá posibilidad de cambiar su estado en el **Bloque de Programa DWH** o en una interrupción (siempre y cuando las interrupciones estén habilitadas).

Sin embargo, la condición solo es evaluada al terminar el **Bloque de Programa DWH**, si en algún momento la condición cambia a falsa, y nuevamente a verdadera, al evaluar la condición solo tomará el valor presente, y nunca valores pasados.

EJEMPLO No. 16

Uso de la sentencia “do...while”

Objetivo:

Escribir una rutina en C que reciba una cadena de caracteres vía serial SCI, mediante el uso de una función interna ya elaborada llamada getchar(), la cual retorna el valor recibido por el SCI del microcontrolador.

El carácter recibido deberá presentarse en uno de los puertos, una vez se reciba el fin de cadena <enter> terminará la recepción de caracteres.

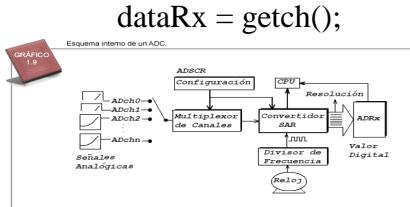
```
***** Ejemplo 16 *****
// Uso de la sentencia do...while
// Fecha: Abril 14,2009
```

```
// Asunto: Recepción de cadena
// de caracteres con ciclo
// Hardware: Sistema de desarrollo PIC-Link Microchip™.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
```

```

#include "dowhile.h"
#define ENTER 0xD // definición de la constante <enter>
#define LNG_MAX 20 // define longitud máxima a recibir
#define Out1_ON() output_high(PIN_B4)
#define Out2_ON() output_high(PIN_B5)
void ShowDataRx(char dato2Show){
    output_d(dato2Show); //presenta dato en el Puerto
}
void Reciba_Cadena(void){
unsigned char i; // offset del buffer de recepción
char dataRx;
    i = 0; // inicializa el valor de i (ppio del buffer)
    do{
        dataRx = getch();
        ShowDataRx(dataRx);
        i++;
    }while( (i != LNG_MAX ) && (dataRx != ENTER));
}
void Mcu_Init_PIC(void){ /*Funcion de Inicializacion*/
    setup_adc_ports(NO_ANALOGS);
    setup_adc(ADC_OFF);
    setup_psp(PSP_DISABLED);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC);
    setup_vref(FALSE);
}
void main(){/*Funcion principal*/
    Mcu_Init_PIC();
    printf(" Ejemplo Ciclo do..while");
    Out1_ON();
    Reciba_Cadena();
    Out2_ON();
    for(;;){
    }
}

```



Discusion:

En el desarrollo de la solución se usa la rutina ShowDataRx (), como función auxiliar para sacar el carácter por el puerto D, el ciclo do..while permanece y valida dos condiciones: una para verificar el número máximo de caracteres a recibir,o bien que el

caracter recibido es ENTER.

Como mínimo el ciclo de ejecuta una vez, y al final verifica las condiciones, las cuales de permanecer falsas, obligan un nuevo ciclo do..while.

6.2.3 La sentencia “while”

La secuencia **while** es muy similar a la do.. while, se utiliza de manera similar en secciones de código que requieren esperar una condición o que se ejecuten cierto número de veces si en la condición se pone un contador. Su diferencia con el do...while radica en que primero se hace la evaluación de la condición y luego la decisión si el bloque de programa se ejecuta o no. Su estructura es:



La sentencia **while** es muy similar a la **do...while**, pero se diferencia en que el **while** exige la comprobación de la condición establecida por el programador antes de ejecutarse.



A

while(<condicion>){

B

Bloque de Programa HW

}

D

Una vez el programa llegue al punto A evaluará la condición, si esta resulta ser verdadera pasará el control al punto B y ejecutará el **Bloque de Programa WH**, terminado nuevamente pasará a evaluar la condición y si ésta persiste verdadera, nuevamente ejecutará el **Bloque de Programa WH**, de esta forma hasta que en la evaluación de la condición resulte ser falsa, en cuyo caso no ejecutará el bloque de programa y saltará directamente al punto D.

Nótese que si al llegar al punto A la condición es falsa la primera vez, no ejecutará el **Bloque de Programa WH** ni una sola vez y pasará al punto D.

EJEMPLO No. 17

Uso de la sentencia “while”

Objetivo:

Escribir una función en C que reciba como argumento una variable de tipo unsigned char, y retorne el valor de su factorial tipo unsigned long.

Usar el ciclo **while** para determinar el valor del resultado.

Solución:

```
***** Ejemplo 17 *****

// Uso de la sentencia “while”

// Fecha: Feb 4,2009

// Asunto: Factorial de un uchar.

// Hardware: Sistema de desarrollo AP-Link

// Freescale™ y PIC-Link Microchip™.

// Versión: 1.0 Por: Gustavo A. Galeano A.

*****



#define __FREESCALE_

#ifndef __PIC__

#define __FREESCALE__

#include <hidef.h> /* for EnableInterrupts macro */

#include “derivative.h” /* include peripheral declarations */

#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP

void Mcu_Init(void){

    Disable_COP();

}

#endif

#define __PIC__

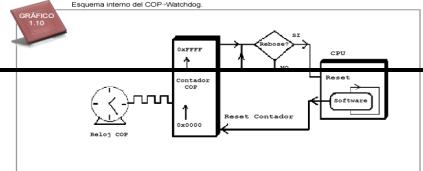
#include “while.h”

typedef int32 ulong;

void Mcu_Init(void){
```

```
    Mcu_Init_PIC(); //función igual a la del Ejemplo No. 16  
}  
  
#endif
```

```
ulong Factorial(unsigned char var){  
  
    ulong result;  
  
    unsigned char num;  
  
    result = 1;  
  
    num = var;  
  
    if(num != 0){  
  
        result = 1;  
  
        while(num != 1){  
  
            result =(unsigned long)num*result;  
  
            num--;  
  
        }  
  
        return result;  
  
    }else{ // caso trivial variable es 0, su factorial es 1  
  
        return 1;  
  
    }  
  
}  
  
ulong factResult;  
  
void main(void) { //Función principal  
  
    Mcu_Init();  
  
    factResult = Factorial(21);
```



```
factResult = Factorial(1);
```

```
for(;;) {
    /* loop forever */
}
```

Discusión:

Usando el ciclo *while* se puede ir decrementando el valor de *num* hasta llegar a 1, e ir calculando finalmente el valor del factorial y retornar su valor.

Una sentencia *if* al comienzo soluciona el caso trivial del valor factorial de 0 y de 1.

Es posible usar la recursividad para el desarrollo del ejemplo y pudiera quedar un poco más eficiente, teniendo en cuenta que $\text{Factorial}(N) = N * \text{Factorial}(N-1)$, sin embargo se debe recordar que en la medida que *N* sea un valor elevado el stack puede desbordarse, debido a que en cada llamado el SP(stack pointer) es decrementado en 2 (dirección de retorno), así que la rutina puede no quedar muy robusta.

Con la sentencia **while** se puede generar un bucle infinito haciendo que la <condición> sea siempre verdadera así:

```
while(1){
```

Bloque de programa

}

El código contenido en el bloque de programa se ejecutará una y otra vez de forma indefinida.

6.2.4 La sentencia “for”

La sentencia **for** es una de las más completas y útiles en la programación estructurada, permite agrupar un código de programa, el cual se ejecuta con base en ciertas condiciones dadas.

Su estructura es:

A

for(<inicializacion>; <condicion>; <incremento>){

B

bloque de programa FR

}

D

En la sentencia **for** existen tres partes principales:

La **<inicializacion>** que solo se ejecuta una vez al iniciar el **for**, normalmente es una sentencia de asignación que se utiliza para inicializar una variable de control del bucle.

La **<condicion>** es una expresión relacional que determina cuando finaliza el bucle.

El **<incremento>** define como cambia la variable de control cada vez que termina el bucle.



La sentencia **for** es una de las más utilizadas en programación de sistemas embebidos, ya que permite agrupar un código de programa bajo ciertas condiciones dadas.



Las tres partes deben estar separadas por punto y coma y todas son opcionales, es decir, que puede existir un **for(;;)**. Sin **<inicializacion>**, sin **<condicion>** en cuyo caso se asumirá **VERDADERA**, y sin **<incremento>**.

La forma de operar es la siguiente:

Una vez el control del programa llega al **punto A**, ejecuta la **<inicializacion>**, a continuación evaluará la **<condicion>**, si ésta resulta ser verdadera ejecuta el **Bloque de Programa FR**, una vez terminado ejecuta el **<incremento>** y evalúa la **<condicion>** nuevamente, si persiste verdadera nuevamente ejecuta el **Bloque de Programa FR** y así sucesivamente hasta que en una evaluación de la **<condicion>** esta resulte ser falsa, en cuyo caso pasará al **punto D** para continuar con otra sentencia.

Variaciones de la Sentencia “for”

La sentencia **for** tiene muchas variaciones y dependiendo del uso que se busque puede tener o no alguna de sus partes, así el bucle infinito de ejecución es el siguiente:

```
for (;;) {
```

bloque de
programa

```
}
```

Este código ejecutará el **Bloque de Programa** una y otra vez, debido a que la condición que pudiera sacarlo no existe, en cuyo caso se define como **VERDADERA** (es equivalente a la secuencia **while(1){ ... }**).

Una variación adicional permite utilizar el separador coma para permitir dos o más variables de control del bucle, así por ejemplo, las variables **i** y **j** controlan el siguiente bucle y ambas son inicializadas dentro de la sentencia **for**.

```
unsigned char i,j; for( i =0, j=0 ; (i+j) < 55 ; i++ ){ ..... // bloque de programa. }
```

Pueden también existir sentencias **for** sin bloque de programa, en algunos casos se utilizan para generar pequeños retardos, como sería:

```
unsigned int i;
```

```
for(i=0; i< 10000; i++);
```

La cual contará de 0 a 10000 sin ejecutar bloque de programa alguno. En otros casos se usan para optimizar procedimientos de búsqueda y de movimiento de apuntadores de forma rápida, así por ejemplo:

```
char Buffer[200] = " Esta es una cadena de caracteres ";
```

```
char *ptr; for(ptr=Buffer; *ptr ; ptr++);
```

Ubicará rápidamente el apuntador ptr al final de la cadena Buffer.

EJEMPLO No. 18

Uso de la sentencia “for”

Objetivo:

Escribir un programa en C que realice el encendido y apagado del led **OUT-1** por 20 veces, una vez que se presiona la tecla **INPUT-1**.

Utilizar para ello una función de retardo **Delay()** entre encendido y apagado, la cual recibe como argumento, el número de iteraciones del retardo.

Solución:

```
***** Ejemplo 18 *****
// Uso de la sentencia “for”
// Fecha: Feb 4,2009
// Asunto: Conteo en variables con ciclo “for”.
// Hardware: Sistema de desarrollo AP-Link
// Freescale™ y PIC-Link Microchip™.
// Versión: 1.0 Por: Gustavo A. Galeano A.
//*****  

#define __FREESCALE_H
#ifndef __PIC_H
#define __FREESCALE_H
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#define DisableWatchdog() CONFIG1_COPD = 1
#define INPUT_1() !PTD_PTD1
#define OUT_1_On() PTC_PTC3 = 1; DDRC_DDRC3 = 1
#define OUT_1_Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1
void Mcu_Init(void){
    DisableWatchdog();
}
#endif
#define __PIC_H
#include "ciclofor.h"
typedef int32 ulong;  

#define INPUT_1() !input(PIN_B0)
#define OUT_1_On() output_high(PIN_B4)
#define OUT_1_Off() output_low(PIN_B4)
```

```

void Mcu_Init(void){
    Mcu_Init_PIC(); //cuerpo en el Ejemplo 4
}
#endif
#define NRO_BLINKS 20
unsigned char nroLeds;
void Delay(ulong nroIteraciones){
    ulong i;
    for(i=0;i<nroIteraciones;i++);
}
void main(void) { //Función principal
    Mcu_Init();

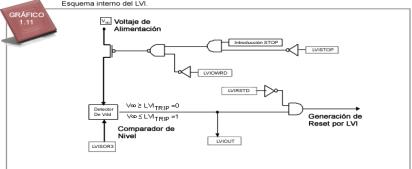
    
    Esquema interno del LVI
      

    Diagrama de circuito que muestra la alimentación (Voltage de Alimentación) conectada a un divisor de tensión (Divisor de Voltaje). El divisor tiene tres salidas: una a un comparador de nivel (Vcc > LM339p > 0), otra a un LED (LUZONDO) y una tercera a un integrador (Integración R/22P). La salida del integrador se conecta a la base de un transistor que controla la salida OUT_1. Una retroalimentación de OUT_1 se conecta al divisor de voltaje. Una tecla INPUT_1 también se conecta al divisor de voltaje. Una señal de generación de复位 (Generación de Reset por LVI) se conecta a la base del transistor.
}

    for(;;) {
        while(!INPUT_1());
        for(nroLeds=0; nroLeds < NRO_BLINKS; nroLeds++){
            OUT_1_On();
            Delay(5000);
            OUT_1_Off();
        }
        Delay(2000);
    }
} /* loop forever */
/* please make sure that you never leave main */
}

```

Discusión:

El ejemplo desarrollado cuenta con 3 ciclos **for**, el que está dentro de la función **Delay()**, es un ciclo **for** sin bloque de programa por ejecutar, el que se encuentra en el programa **main()**, **for(;;)**, implementa un ciclo sin **<inicialización>**, sin **<condición>** y sin **<incremento>**, el cual describe un ciclo infinito, y el tercero y más completo el usado para contabilizar los ciclos que debe encender y apagar el led, dada la presión de la tecla **INPUT-1**.

6.2.5 La Sentencia “switch”



Con la sentencia **switch** el programador está en capacidad de subdividir un módulo

La sentencia **switch** es tal vez una de las más importantes y de mayor uso en los proyectos tratados más adelante, debido a la estructura vertical que tiene y su capacidad de poder dividir un módulo en estados.

La sentencia **switch** nace como una forma sencilla de tener if anidados, los cuales pueden ser un poco confusos cuando se habla de más de 3 if anidados, entre los cuales deberá existir por lo menos un tabulador, haciendo que el código se extienda de manera horizontal lo que dificulta su lectura y entendimiento.

La estructura de la sentencia “**switch**” es:

A

switch(<variable>){

case VALOR _1:

bloque de programa V_1

break;

case VALOR _2:

bloque de programa V_2

break;

case VALOR _3:

bloque de programa V_3

break;

.....

case VALOR _N:

bloque de programa V_N

break;

default:

en estados de manera sencilla. Es un proceso que también se podría hacer por la sentencia if, pero ocurre que más de 3 if anidados genera confusión, lo que resuelve el switch.



```

        bloque de programa V_DF

break;
}

D

```

Al llegar el control del programa al punto A, toma la variable y la compara con el **VALOR _1**, si ambos valores son diferentes, pasará a comparar el valor de la variables con el del case **VALOR _2**, **VALOR _3**, hasta que encuentre coincidencia; en el momento que encuentre igualdad, iniciará la ejecución del **bloque de programa V_N** que coincide hasta que encuentre la palabra reservada **break**, en cuyo caso se irá directo al **punto D** terminando la sentencia **switch**.

En teoría, se pueden tener tantos “cases” como se quiera o soporte el tipo de la <variable> en cuestión, si al llegar al **switch**, la variable NO coincide con ninguno de los **VALOR _1 ... VALOR _N**, se ejecutará **el bloque de programa V_Df como bloque “default”**.

La estructura **switch** solo exige el **switch** y un solo case, tanto la palabra “**break**”, como la palabra “**default**” son opcionales para que no genere error de sintaxis, sin embargo, exige especial atención cuando se suprimen estas palabras.

La sentencia **switch** entonces puede tener las siguientes variaciones, dependiendo de la presencia de las palabras “**break**” y “**default**”. Consideremos esta estructura:

A

```

switch(<variable>){

    case VALOR _1:
        bloque de programa V_1
        break;

    case VALOR _2:
    case VALOR _3:
        bloque de programa V_2_3
        break;
}

```

default:

bloque de programa V_DF

break;

case **VALOR _N**:

bloque de programa V_N

break;

default:

bloque de programa V_DF

break;

}

D

En este caso el programa ejecutará el mismo código si el valor de la variable es **VALOR _2** ó **VALOR _3**, con lo cual se optimiza memoria de programa ante dos o más procedimientos que ejecuten el mismo código ante su igualdad.

También se puede ejecutar un código previo cuando la igualdad tenga un valor diferente y continuar con el código común a otro bloque.



El hecho que no se requieran los **break** y los **default** en la sentencia **switch** no indica que se deban omitir desde un principio, en su lugar la sentencia se debe iniciar con todos los **break** y la sección **default** tal como se hizo la definición al principio, si después de evaluar el código se encuentran valores redundantes o que realizan el mismo código, se procede a juntarlos y eliminar el **break** que los separa y de esta forma optimizar código, o si algunas variables tienen un código en común se mueven a un lugar cercano y solo se le adiciona el código previo a una de ellas.

De igual forma si el **switch** no lleva **default** por no requerirse o porque resulta ser redundante, se remueve solo al final de forma consciente por el programador.

EJEMPLO No. 19

Uso de la sentencia “switch”

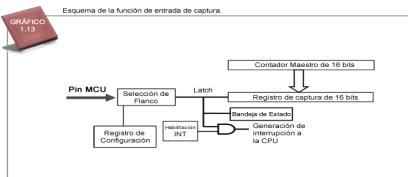
Objetivo:

Escribir un programa en C que realiza diferentes acciones, dependiendo de la tecla presionada.

A continuación se describen las teclas, y su acción respectiva.

NOMBRE TECLA	ACCIÓN
KEY_ENTER (INPUT_1)	Enciende temporalmente OUT_1
KEY_ESCAPE (INPUT_2)	Enciende temporalmente OUT_1 y BUZZ
KEY_INC	Enciende OUT_2
KEY_DEC	Apaga OUT_2

Las teclas tienen la prelación listada en la tabla anterior. **Solución:** //***** Ejemplo 19 ***** // Uso de la sentencia “switch” // Fecha: Feb 4,2009 // Asunto: Acciones de cada tecla. // Hardware: Sistema de desarrollo AP-Link(2008-01-14) // para Microcontrolador Freescale™ AP16. // Versión: 1.0 Por: Gustavo A. Galeano A. //***** #define __FREESCALE__ //define __PIC__ #ifdef __FREESCALE__ #include <hidef.h> /* for EnableInterrupts macro */ #include "derivative.h" /* include peripheral declarations */ #define DisableWatchdog() CONFIG1_COPD = 1 void Mcu_Init(void){ DisableWatchdog(); } #define INPUT_1() !PTD_PTD1 #define INPUT_2() !PTD_PTD2 #define INPUT_3() !PTD_PTD3 #define INPUT_4() !PTD_PTD4 #define OUT_1_On() PTC_PTC3 = 1; DDRC_DDRC3 = 1 #define OUT_1_Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1 #define OUT_2_On() PTC_PTC2 = 1; DDRC_DDRC2 = 1 #define OUT_2_Off() PTC_PTC2 = 0; DDRC_DDRC2 = 1 #define Buzz_On() PTD_PTD0 = 1; DDRD_DDRD0 = 1 #define Buzz_Off() PTD_PTD0 = 0; DDRD_DDRD0 = 1 #endif #ifdef __PIC__ #include "sentswitch.h" typedef int32 ulong; void Mcu_Init(void){ Mcu_Init_PIC(); /*cuerpo de la función en el Ejemplo 4*/ } #define INPUT_1() !input(PIN_B0) #define INPUT_2() !input(PIN_B1) #define INPUT_3() !input(PIN_D2) #define INPUT_4() !input(PIN_D1) #define OUT_1_On() output_high(PIN_B4) #define OUT_1_Off() output_low(PIN_B4) #define OUT_2_On() output_high(PIN_B5) #define OUT_2_Off()



```

output_low(PIN_B5) #define Buzz_On() output_high(PIN_B2)
#define Buzz_Off() output_low(PIN_B2) #endif void Delay(ulong
nroIteraciones){ ulong i; for(i=0;i<nroIteraciones;i++); } #define
KEY_ENTER 1 #define KEY_ESC 2 #define KEY_INC 3 #define
KEY_DEC 4 #define KEY_INVAL 5 void Key_Init(void){} char

```

Key_Read(void){ if(INPUT_1()) return KEY_ENTER; if(INPUT_2()) return KEY_ESC; if(INPUT_3()) return KEY_INC; if(INPUT_4()) return KEY_DEC; return KEY_INVAL; } void Key_Run(void){ char nroTecla; nroTecla = Key_Read(); switch(nroTecla){ case KEY_ESC: Buzz_On(); case KEY_ENTER: OUT_1_On(); break; case KEY_INC: OUT_2_On(); break; case KEY_DEC: OUT_2_Off(); break; default: break; } Delay(3000); OUT_1_Off(); Buzz_Off(); Delay(3000); } void main(void) { //Función principal Mcu_Init(); Key_Init(); for(;;) { Key_Run(); } /* loop forever */ } **Discusión:** La solución al ejercicio bien pudiera hacerse con varias secuencias if; sin embargo, la estructura con la secuencia **switch** es mucho más clara. Nótese que la acción de la tecla **KEY_ESC** realiza, además de encender el **BUZZER**, la misma acción que la tecla **KEY_ENTER**, con lo que su **break** correspondiente se puede omitir. La prueba completa del ejemplo usando la tarjeta, requiere que se realice el montaje de las teclas **KEY_INC** y **KEY_DEC** en el conector respectivo. La sección **default**, en esta caso resulta opcional debido a que no realiza acción ninguna, sin embargo se deja para ilustrar la estructura completa de la sentencia **switch**. Es muy posible que el compilador al realizar su proceso optimice el código de tal forma que elimina la sección **default**, sin embargo resulta útil dejarla en los programas desarrollados para notificar errores en los cuales la variable del **switch** no resulta igual a ninguno de los valores, con esta notificación el programa puede tener diferentes acciones correctivas, como inicializar variables o notificarlo a otros módulos de alto nivel. En el caso del ejemplo puede usarse para saber que una tecla fue presionada pero no es ninguna tecla válida especificada para realizar alguna acción. También y dado que en este caso los bloques de programa son cortos (de una sola línea), se pueden incorporar en la misma línea del caso, pero si en algún caso se requiere que el bloque de programa sea de más líneas, se recomienda que el bloque de programa respectivo al caso, arranque en una línea inferior.

Contrario a lo que se cree el **switch** puede resultar ineficiente cuando el valor de la variable coincide con el último valor, ya que se puede tener la tendencia a pensar que el procesador deberá hacer muchas comparaciones antes de llegar al valor de igualdad. Sin embargo, compiladores tan óptimos como el Codewarrior®, generan una tabla de saltos que harán que cualquier valor que tome la variable se encontrará tan rápido como si fuera el primer valor; por esta razón no importará el orden en el cual se posic和平en los valores a evaluar en cada **case**, ya que en tiempo de ejecución se tardará lo mismo para encontrar el primer o el último valor.

6.2.6 La sentencia “break”



La
break
se
utiliza

Esta sentencia tiene dos usos: como se vio anteriormente para delimitar un **case** de una sentencia **switch**, y también se puede usar para **FORZAR** la terminación inmediata de un bucle saltando la evaluación condicional normal del ciclo.

En el siguiente código:

```
unsigned int contador; signed svble1;  
for(contador = 0; contador < 100; contador++){  
    svble1++;  
    if(contador == 20 ) break;  
}  
D
```

en situaciones en las cuales un bucle puede ser interrumpido por una condición especial, así la evaluación de la condición normal del ciclo no haya terminado.



La variable contador que es inicializada en cero, al llegar a 20 suspenderá y saldrá de la secuencia **for** directamente al punto D, inclusive con la condición (contador < 100) verdadera.

La sentencia **break** no es muy frecuente, pero se usa en bucles donde una condición especial puede dar lugar a la terminación inmediata.

Como aclaración final, el **break** solo da lugar a la salida del bucle mas interno, así por ejemplo:

```
for(t=0; t<100; t++ ){  
    cont = 1;  
    for(;;){  
        cont++;  
        if( cont == 10 ) break; /* aborta un nivel de for*/  
    }  
}
```

Solo sale del **for** mas interno, continuando con el conteo de la variable **t** de forma definida en la secuencia **for**.

6.2.7 La sentencia “continue”

La sentencia **continue** funciona de una forma similar al **break**. Sin embargo, en lugar de forzar la terminación, **continue** fuerza una nueva iteración del bucle y salta cualquier código que exista hacia abajo.

Considerando el siguiente código:

```
unsigned char contador;  
unsigned char i,j;  
for(i=50, j=20, contador=0 ; contador<50; contador++){  
    if(contador>10){  
        continue;  
    }  
    i++;  
    j--;  
}
```

Las primeras 10 iteraciones se hacen de forma que i se incrementa y j se decrementa en 1, sin embargo, al llegar contador a 11, no ejecutará el código que continúa en su parte inferior y realizará una nueva iteración del **for**.

Para los bucles **while** y **do... while**, una sentencia **continue** hace que el control del programa pase a la prueba del condicional y siga con el proceso de iteración.

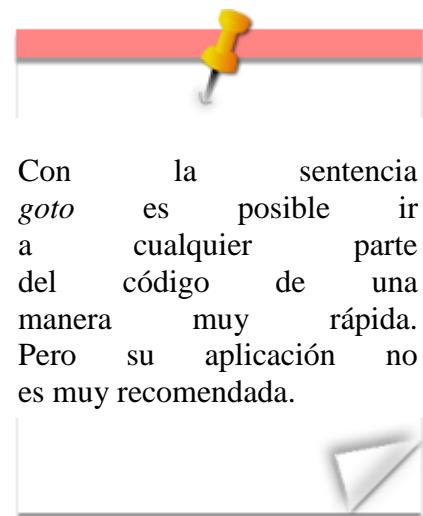


La sentencia **continue** aborta el ciclo interno actual de ejecución, y obliga a que el código ejecutado sea el que se encuentra en la parte inferior y exterior al ciclo.



6.2.8 La Sentencia “goto”

Con la sentencia **goto** es posible ir a cualquier parte del código de una manera muy rápida. Pero su aplicación no es muy recomendada.



El uso de esta secuencia ha decaído mucho en los últimos años, debido a que hacen los programas un poco ilegibles; sin embargo, se muestra su uso ya que en algunos casos pudiera el programador usarlo y es útil saber en ese caso como hacerlo. Por esta razón el **goto** no se usa en esta publicación fuera de esta sección, también debido a que el C cuenta un gran conjunto de estructuras de control adicionales, como el **break** y el **continue**, de tal forma que su uso no es indispensable.

Se ha notado que su uso es útil en algunos casos donde la velocidad de ejecución es crítica, debido a que puede, con determinada evaluación, ir directamente a algún lugar del programa (adelante o atrás) con mucha velocidad y obviando evaluaciones intermedias.

La sentencia **goto** requiere de una etiqueta la cual no es más que un marcador dentro del código, el cual define una posición del **PC** (*program counter*) dentro del mapa de memoria del procesador. Esta sentencia es equivalente a un BRA (salto sencillo) o un JMP (salto largo) en lenguaje ensamblador, con la diferencia que en C la etiqueta deberá estar en la misma función en la que se utiliza el **goto**.

Ejemplo de su uso:

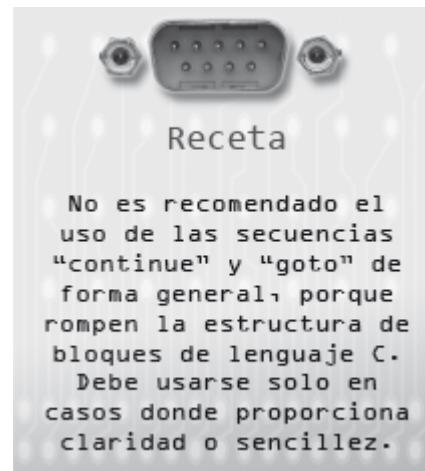
```
unsigned char x;
```

```
x = 1;
```

Etiqueta:

```
x++;
```

```
if(x < 50 ) goto Etiqueta;
```



No es recomendado el uso de las secuencias "continue" y "goto" de forma general, porque rompen la estructura de bloques de lenguaje C. Debe usarse solo en casos donde proporciona claridad o sencillez.

Se debe usar la secuencia **goto** con moderación, si es que se usa. Pero si el código va a ser más difícil de leer mejor no usarlo.

6.3 MESCLA DE C CON LENGUAJE ENSAMBLADOR EN SISTEMAS EMBEBIDOS

La inclusión de lenguaje ensamblador dentro de un proyecto desarrollado en C puede ser necesaria por varias razones:

Ejecutar procedimientos muy optimizados en velocidad de ejecución: como es sabido el compilador puede generar más código del necesario, y aunque esto no debería ser crítico para el desarrollo del proyecto, existen secciones que sí pueden requerir un procedimiento muy veloz.

Manejar alguna característica de un microcontrolador que por medio de lenguaje C no sea posible: así por ejemplo, en el procesador AP16A existe la instrucción BIH (*branch if IRQ High*) y BIL (*branch if IRQ Low*), las cuales realizan lectura directa del pin IRQ y toman una decisión de salto. No existe en C una directiva o sentencia propia que lo realice, y en este caso es necesario hacerlo por la vía del lenguaje ensamblador.

Invocar instrucciones específicas de ensamblador: instrucciones para invocar el bajo consumo de energía (STOP, WAIT) o invocar una interrupción de software (SWI).

Generar pequeños lapsos de temporización muy breves: en C la medida de tiempo por ejecución de código no existe, porque las instrucciones generadas pueden variar, dependiendo del compilador, del tipo de optimización etc; mientras que en código de máquina, y por el hecho de estar intacta su generación, se puede medir de forma precisa el tiempo que un procedimiento se tarda en ejecutar.

El código en ensamblador escrito no será optimizado (haga lo que haga), y tampoco verifica si interfiere con el código en C; al mismo tiempo, el código ensamblador puede interactuar con las variables (leerlas y/o modificarlas) y funciones declaradas en el proyecto.



El programador debe tomar todas las precauciones cuando inserte lenguaje ensamblador dentro del código C, porque algunas operaciones podrían presentar un comportamiento distinto al esperado.



Al incluir el código deberá ponerse especial énfasis en el manejo del *stack pointer* (SP), como se sabe es uno solo y su alteración podría provocar malfuncionamiento en el proyecto una vez se abandone la sección, de tal forma que debe garantizarse la entrega del SP en la misma dirección en la que fue recibido al iniciar la sección en ensamblador; tener en cuenta que instrucciones que envíen datos al *stack* (ejemplo: PSHA, PSHX ..) alteran la posición del *stack pointer*, por lo que deben tener su correspondiente instrucción paralela (ejemplo: PULA, PULX ...), al terminar el procedimiento. El programador debe tomar todas las precauciones cuando inserte lenguaje ensamblador dentro del código C, porque algunas operaciones podrían presentar un comportamiento distinto al esperado.

De igual forma, pudieran usarse direccionamientos orientados al *stack*, sin modificar los valores que están debajo del *stack* (direcciones mayores con la que se recibió el SP), allí están almacenadas variables propias del C que están siendo usadas por el código.

Existen tres formas de incluir lenguaje ensamblador dentro de un proyecto en C, y su uso depende de la cantidad de código que se requiera insertar:

Código en Línea 1: usado cuando se requiere insertar pocas líneas de lenguaje ensamblador, (normalmente una línea), y tiene la siguiente sintaxis:

```
asm("<instrucción en ensamblador>");
```

Donde:

<instrucción en ensamblador> es cualquier instrucción del procesador usado con su respectivo modo de direccionamiento.

Código en Línea 2: usando cuando se requiere insertar un código intermedio de lenguaje ensamblador. La sintaxis es:

```
__asm <instrucción # 1 en ensamblador>
__asm <instrucción # 1 en ensamblador>
__asm <instrucción # 1 en ensamblador>
```

```
.....  
_asm <instrucción # n en ensamblador>
```

Donde:

<instrucción # i en ensamblador> es una instrucción completa con su respectivo direccionamiento.

Bloque de ensamblador: se usa cuando se requiere insertar un bloque de lenguaje ensamblador contenido varias instrucciones secuenciales.

Esta forma es la recomendada porque es la más popularizada en los compiladores comerciales. Su sintaxis es la siguiente:

```
#asm  
  
<instrucción # 1 en ensamblador>  
  
<instrucción # 2 en ensamblador>  
  
<instrucción # 3 en ensamblador>  
  
....  
  
instrucción # n en ensamblador>  
  
#endasm
```

Donde:

<instrucción # i en ensamblador> es una instrucción completa con su respectivo direccionamiento.

#asm y #endasm: determinan respectivamente el inicio y fin de la sección en código ensamblador, ambos deben iniciar en la columna 1 del editor.

Como se puede notar la forma 1 será la indicada cuando se van a insertar una o dos líneas de ensamblador, permite la tabulación de acuerdo con el flujo del programa, mientras que la forma 3 exige que la primera y última línea del bloque deban empezar en la columna 1, lo que viola la tabulación propia del archivo sobre el cual se está trabajando y por esto es mejor y recomendable hacer una rutina que sea invocada desde el C y en esta nueva función va el nuevo código en ensamblador.

Este formato es aceptado por los dos compiladores que se muestran en esta publicación, el CCS y el Codewarrior ® y es tal vez uno de los más generales y estándar.

EJEMPLO No. 20

Mezclando lenguaje C y ensamblador

Objetivo:

Escribir las siguientes rutinas en ensamblador sobre un proyecto en C:

1. Rutina GetCCR() la cual una vez invocada, retorne en un char, el estado actual del CCR (*Condition Code Register*) .
2. Rutina de retardo corto que retarda tantos ciclos de máquina definidos en un macro tipo unsigned int (valor de 16 bits).

Realizar un programa que pruebe el resultado entregado por GetCCR(), que indique con un encendido temporal el estado de la bandera I del CCR en la salida OUT_1 si la bandera esta activa y en OUT_2 si la bandera está inactiva, la duración de la indicación tendrá la duración de la rutina Delay() .

3. Realizar una rutina que calcule la suma (checksum) de los datos almacenados en toda la memoria flash del microcontrolador AP16A.

Solución:

```
***** Ejemplo 20 *****  
//  
// Mezclando lenguaje C y ensamblador  
//  
// Fecha: Feb 4,2009  
// Asunto: Uso de lenguaje de maquina en C.  
//  
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)  
// para Microcontrolador Freescale AP16.  
// Version: 1.0 Por: Gustavo A. Galeano A.  
//  
*****  
  
#include <hidef.h> /* for EnableInterrupts macro */  
#include "derivative.h" /*include peripheral declarations*/  
  
// declaracion de macros  
  
#define DisableWatchdog() CONFIG1_COPD = 1  
#define OUT_1_On() PTC_PTC3 = 1; DDRC_DDRC3 = 1  
#define OUT_1_Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1  
#define OUT_2_On() PTC_PTC2 = 1; DDRC_DDRC2 = 1  
#define OUT_2_Off() PTC_PTC2 = 0; DDRC_DDRC2 = 1
```

```

#define NRO_DLY 62000

#define FLASH_INI 0x0860
#define FLASH_END 0x485F

// declaracion de variables globales

char ccr;
volatile char estadoCCR;
volatile char checkSum@0x0080; //variable directa en 0x0080

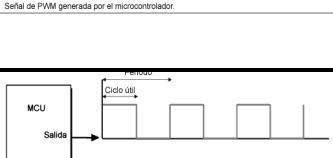
char GetCCR(void){
__asm psha
__asm tpa
__asm sta ccr
__asm pula
return ccr;
}

// declaracion de rutina de retardo
void Delay(void){
#asm
    ldhx #NRO_DLY
loopDly: aix #-1
    cphx #0
    bne loopDly
#endasm
}

// Rutina de cálculo de CheckSum
void CheckSumAP16A(void){
#asm
    clr checkSum
    ldhx #FLASH_INI
loopChk: lda 0,x
    add checkSum
    sta checkSum
    aix #1
    cphx #FLASH_END+1
    bne loopChk
#endasm
}

void main(void) {
    DisableWatchdog();
}

```



```
EnableInterrupts; /* enable interrupts */
/* include your code here */
```

```
estadoCCR = GetCCR();
if(estadoCCR & 0b0001000){ // verifica estado de la bandera I
    OUT_1_On();
    Delay();
    OUT_1_Off();
} else{
    OUT_2_On();
    Delay();
    OUT_2_Off();
}

CheckSumAP16A();

for(;; {
} /* loop forever */
/* please make sure that you never leave main */
}
```

Discusión:

Para la implementación de la rutina **GetCCR()** se requiere usar lenguaje ensamblador debido a que no hay una función o sentencia que permita obtener el registro de banderas **CCR** de la máquina, al llegar a la rutina se guarda el contenido del acumulador **ACCA**, el cual no modifica el estado del **CCR**, se transfiere luego el valor del **CCR** al **ACCA** usando la instrucción **TPA** y se almacena en la variable que va a ser retornada.

En esta rutina se usa la opción 2 debido a que son pocas instrucciones.

Para la implementación de la rutina **CheckSumAP16A()** se usa la opción 3 para ilustrar su uso y debido a que la rutina interactúa de forma global.

Se carga el valor inicial de la memoria flash en el registro **H:X**, se direcciona el valor almacenado en esta dirección y se suma a la variable **checkSum**, procedimiento que se hace hasta que la dirección almacenada en el **H:X** sea la dirección final de la memoria.

6.4 RECURSIVIDAD EN SISTEMAS EMBEBIDOS

En C, las funciones pueden llamarse a sí mismas, de forma similar a como se haría en lenguaje ensamblador así:

Inicio

```
lda # 5 jsr Subrutina1 ; realiza primer llamado a Subrutina1
```

Subrutina1

```
deca           ; decrementa AccA
```

```
cmpa # 0      ; compara con cero
```

```
beq outSub    ; es Z = 1 (cero)?
```

```
jsr Subrutina1 ; sino realiza llamado a si misma
```

outSub

```
RTS
```

Si una función realiza un llamado a la propia función que se está ejecutando, se dice que esta es una función recursiva. La recursividad es el proceso de definir algo en términos de sí mismo y algunas veces se llama definición circular.

Se debe tener en cuenta que cada llamado recursivo a sí misma, genera que el *stack* disminuya como mínimo en 2, más el número de bytes de los argumentos que requiere la función, y por esto deberá garantizarse que ante la peor situación el stack no se desbordará, lo que puede ocasionar daño en las variables locales y posiblemente el programa no retorne de nuevo al punto donde la función fue llamada por primera vez.

El ejemplo más popular de recursividad es la *función factorial*, la cual por definición para un valor N es el producto de $N*(N-1)*(N-2)* \dots *2*1$.



La recursividad en los sistemas embebidos hace referencia a la posibilidad que una función haga un llamado a sí misma, y se usan para crear versiones más sencillas de algoritmos complejos.

```

unsigned int Factorial(unsigned int dato){

    unsigned int resultado;
    resultado = 1;

    if((dato == 0) || (dato == 1)) return resultado; //

    /*resuelve casos triviales*/

    resultado = dato*Factorial(dato-1); // realiza llamado Recursivo

    return resultado;

}

```

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claros y más sencillos que sus equivalentes iterativas.

Algunos problemas, especialmente los relacionados con la inteligencia artificial, parece que tienden ellos mismos hacia soluciones recursivas. También pensando en el programador, parece que algunos piensan más fácilmente de forma recursiva que de forma iterativa.

6.5 ARREGLOS (ARRAYS) DE DATOS

Un arreglo de datos es una secuencia de variables del mismo tipo que se denominan por un nombre común.

A un elemento específico de un arreglo de datos se accede mediante un índice. En C todos los arreglos de datos constan de posiciones de memoria continuas. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento. Los arreglos de datos pueden tener una o varias dimensiones.



6.5.1 Arreglos unidimensionales

La forma general de declaración de un arreglo de datos unidimensional es:

```
tipo nombre_Array[tamano];
```

En donde:

tipo: define el tipo de variable de los elementos del *array*.

nombre_Array: define el nombre del *array*.

tamano: define el número de elementos del *array*.

Todos los arreglos de datos tienen el 0 (cero) como índice de su primer elemento, por lo tanto al último elemento se accederá mediante el índice (tamano-1).

Así por ejemplo:

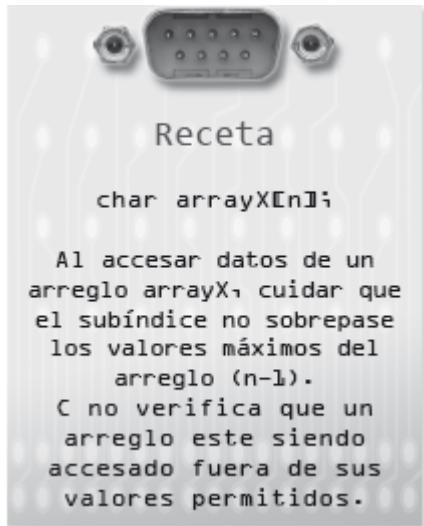
```
char array1[12];
```

Esta declarando un arreglo de datos o de caracteres que tiene 12 elementos de tipo char, cuyos 12 elementos individuales son accesados así:

array1[0] → para acceder al primer elemento.

...

array1[11] → para acceder al último elemento.



Los arreglos de datos son útiles en programación para agrupar sobre una única entidad un grupo de valores del mismo tipo, sea una tabla de datos, una secuencia o una matriz de valores.



Debido a que el C NO comprueba los límites de los arreglos de datos, se puede sobrepasar cualquier extremo de un arreglo de datos y acceder por equivocación a valores fuera del arreglo de datos, que puede corresponder a otra variable de datos; para el caso del arreglo de datos declarado anteriormente, el C no generará error en tiempo de compilación si encuentra que se está accediendo elementos por fuera del arreglo de datos declarado, así por ejemplo:

```
array1[20] = 0; /* asigna un cero nueve bytes por fuera del array1 */
```

De igual forma:

```
array1[i] = 0; /* asigna un cero en el elemento i+1 del array1 */
```

El compilador no comprobará que el valor de *i* sea menor que 12, para garantizar que el elemento esta dentro del arreglo de datos declarado, esta responsabilidad se deja al programador quien deberá

garantizar que en el caso extremo no se sobrepasa el acceso al arreglo de datos mismo o que por error se hagan accesos fuera de el mismo.

nombre_Array es un apuntador al primer elemento del array y se denomina el apuntador al arreglo de datos:

Es decir que:

`nombre_Array[0] = nombre_Array`

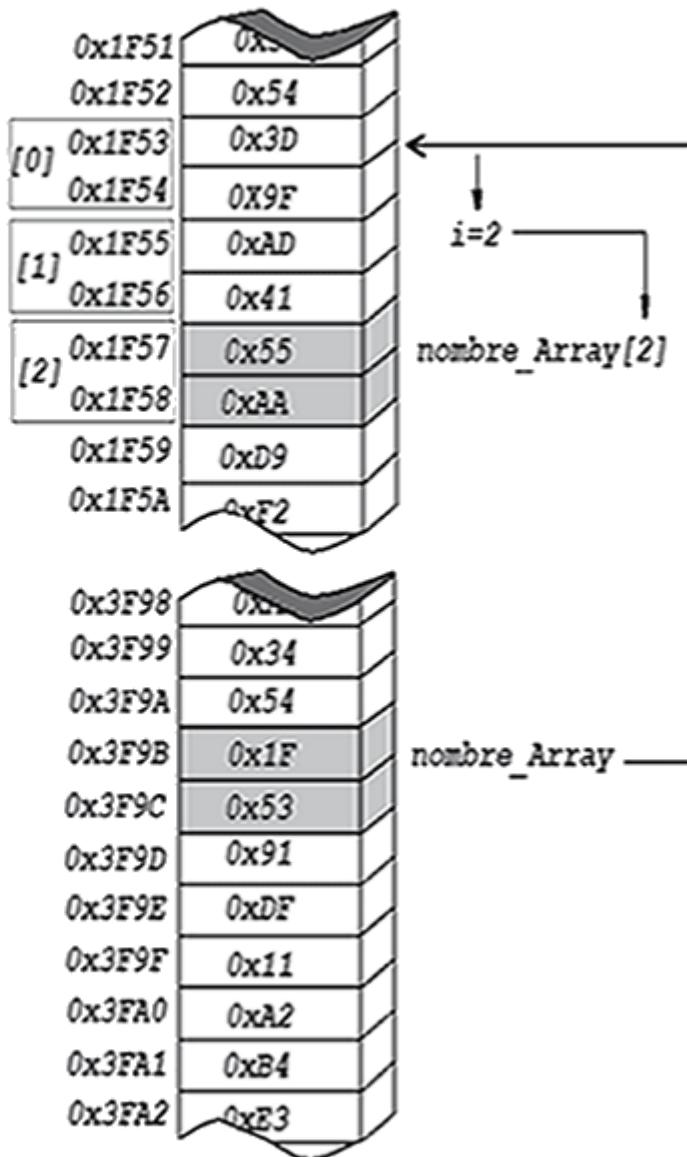
El acceso a cada uno de los componentes del arreglo se puede hacer además tomando el apuntador al arreglo de datos `nombre_Array`, desplazándolo en memoria a la dirección del componente y tomando el valor que se encuentra almacenado en ese apuntador, o lo que es lo mismo:

`nombre_Array[i] = *(nombre_Array + i)`

**GRÁFICO
6.1**

Direccionamiento de componentes dentro un arreglo de datos.

```
uint *nombre_Array;  
nombre_Array = 0x1F53;  
i=2;  
nombre_Array[i]=0x55AA;
```



La forma de la derecha (con subíndice) es mucho más común y sencilla de acceder los componentes, sin embargo, su forma correspondiente de la derecha resulta útil cuando se va a desplazar a través del arreglo en pasos diferentes al valor declarado original.

EJEMPLO No.21**Acceso a arreglos de datos****Objetivo:**

Realizar una función en C que recibe como argumentos el apuntador a un arreglo de datos tipo int, y su tamaño, y los organiza de menor a mayor valor, en la misma estructura que recibe si la tecla **INPUT_1** está presionada, de lo contrario organiza de mayor a menor.

Solución:

```
***** Ejemplo 21 *****
// Acceso a arreglos de datos
// Fecha: Feb 4,2009
// Asunto: Ordenamiento de datos.
// Hardware: Sistema de desarrollo AP-Link
// Freescale™ y PIC-Link Microchip™.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#define __FREESCALE_
#ifndef __PIC_
#define __FREESCALE_
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#define DisableWatchdog() CONFIG1_COPD = 1
#define INPUT_1() !PTD_PTD1
#define INPUT_2() !PTD_PTD2
#define Mcu_Init() DisableWatchdog()
#endif
#ifndef __PIC_
#include "arreglosData.h"
typedef int16 uint;
typedef unsigned char uchar;
void Mcu_Init(void){
    Mcu_Init_PIC(); /*cuerpo de la funcion en el Ejemplo 4*/
}
#define INPUT_1() !input(PIN_B0)
#define INPUT_2() !input(PIN_B1)
#endif
#define TAMANO_ARRAY 12
#define MENOR_MAYOR 0
#define MAYOR_MENOR 1

uint arrayDatos[TAMANO_ARRAY]={
450,12500,5,7250,21,11250,35720,120,9000,10,3456,1200
};

/*arrayPtr: apuntador al array a organizar
tamano: numero de datos a organizar
tSort: toma los valores MENOR_MAYOR o MAYOR_MENOR para indicar el
tipo de ordenamiento*/
```

```

void BubbleSort(uint *arrayPtr, uchar tamano, char tSort){
    char intercambio;
    uint dataBck;
    uchar i;
    intercambio = TRUE;
    while(intercambio){
        intercambio = FALSE;
        for(i=0;i<=(tamano-2);i++){
            if((tSort == MENOR_MAYOR) && (arrayPtr[i] > arrayPtr[i+1])
                || ((tSort == MAYOR_MENOR) && (arrayPtr[i] < arrayPtr[i+1]))){
                dataBck = arrayPtr[i];
                arrayPtr[i] = arrayPtr[i+1];
                arrayPtr[i+1] = dataBck;
                intercambio = TRUE;
            }
        }
    }
}

void main(void) { //Función principal
    Mcu_Init();
    if(INPUT_1()){
        BubbleSort(arrayDatos,TAMANO_ARRAY,MENOR_MAYOR);
    }else{
        BubbleSort(arrayDatos,TAMANO_ARRAY,MAYOR_MENOR);
    }
    for(;;){ /* loop forever */
}
}

```

Discución:

Como se puede observar la función recibe la dirección inicial del arreglo por referencia. Y para la organización se acude a la teoría numérica y el algoritmo de ordenamiento denominado de burbuja, que provee una forma sencilla de ir moviendo el dato de menor valor a la izquierda del arreglo y los datos mayores a la derecha.

Es necesario revisar varias veces toda la lista hasta que no se encuentren intercambios, lo cual significa que la lista está ordenada. Este método es también conocido como el método del intercambio directo.

El procedimiento en pseudo-código se realiza de la siguiente forma para organizar de menor a mayor:

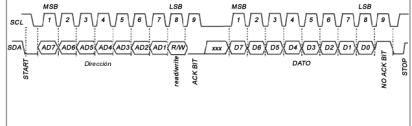
Procedimiento($a_0, a_1, a_2, \dots, a_{n-1}$)

Mientras

Intercambio \leftarrow falso

Para $i=0$ hasta $i=n-2$ haga:

Si ($a_i > a_{i+1}$) entonces $(a_i, a_{i+1}) \leftarrow (a_{i+1}, a_i)$



Intercambio ← verdadero

Si la organización es de mayor a menor, basta con cambiar la comparación > (mayor que) a < (menor que)

En este caso arrayDatos que es el nombre del arreglo es un apuntador al arreglo mismo, o lo que es lo mismo un apuntador al primer carácter del arreglo (arrayDatos es equivalente a &arrayDatos[0]), de esta forma y debe ser claro que a la función BubbleSort() a organizar los datos, se le pasa una dirección (2 bytes) en lugar de todos los valores del arreglo.

La función interna funcionará de igual forma independiente si la tabla es muy corta o muy extensa, y también si el tipo de datos de la tabla cambia a tipos más pequeños como es el char o más grandes como es el unsigned long.

La función BubbleSort() puede ser robustecida para que valide si el segundo argumento llamado tamaño, es 0 o 1 y retorne de inmediato sin realizar ninguna optimización, basta con agregar una línea al principio de la función así:

```
if(tamaño < 2) return;
```

Una forma alternativa de direccionar los datos con el subíndice es hacerlo sumando un offset al apuntador inicial y luego tomar su contenido, en este caso debe recordarse que:

arrayDatos[i] es equivalente a *(arrayDatos + i)

6.5.2 Cadenas de datos “strings”

Un caso muy frecuente de uso de arreglos de datos unidimensionales es una secuencia de caracteres de datos ascii, comúnmente llamado *string* o cadena. El fin de la cadena lo delimita el carácter nulo (0x00), este carácter ocupa un byte al final de la cadena, por lo que cualquier declaración de una cadena, deberá tener capacidad de un carácter adicional para almacenar el fin de *string*.

La declaración de una cadena se hace de la siguiente forma general:

```
char nombre_string[tamaño]=“mensaje inicial”;
```

Donde:

nombre_string: nombre de la cadena.

tamaño: [opcional] longitud de la cadena (incluyendo el carácter de fin de cadena nulo).

“mensaje inicial”: [opcional] define los caracteres iniciales de la cadena.

De forma similar a la declaración de los tipos de variable, la declaración de una cadena acepta todos los modificadores a los tipos de variable.

En el caso de usar el modificador **const**, se define la cadena en memoria de programa o flash.

En este caso la memoria RAM no es involucrada en la declaración y todo el tratamiento de la cadena se hace en memoria de programa, con economía de la memoria RAM; debe recordarse que la cadena, por ser de constantes, exige tener el campo de inicialización: “**mensaje inicial**”, y la cadena podrá ser leída, mas no se puede modificar ninguno de sus componentes.

Cualquier cadena definida dentro de las comillas “”, define un apuntador a esa cadena en memoria de programa, y por esto el acceso a la cadena se hace invocando la cadena completa “cadena de caracteres” o podrá accederse a cualquier o alguno de sus campos usando los subíndices: “**cadena de caracteres**”[2] , dará como resultado ‘d’.

Los campos **tamano** y “**mensaje inicial**” son opcionales, pero es obligatorio uno de los dos. Normalmente, al momento de la declaración se especifica su tamaño máximo, y solo en casos donde se inicializa la cadena el tamaño puede obviarse y reemplazarse solo por los corchetes [].

La cadena puede inicializarse para que al momento de su declaración tenga datos iniciales, esta igualdad es opcional, en este caso el carácter nulo es adicionado al final de las comillas por el C, y no requiere ser adicionada de forma explícita.

En caso de no tener la inicialización, la cadena tendrá ceros en todos sus campos si la declaración es global o **static**, o tendrá datos aleatorios si el string es declarado de forma local dentro de una función.

Al manipular cadenas, el programador debe garantizar que está delimitada por el carácter nulo, las funciones implementadas por otros programadores y por las librerías del C, asumen que existe este terminador para todas sus operaciones internas.

El C en su liberaría **<string.h>**, contiene gran cantidad de funciones para manipular las cadenas, todas con sus prototipos en el archivo de cabecera **<string.h>**, funciones de comparación, copia, tamaño, entre otras. Estas funciones estan listadas en el Anexo # 2.

Ejemplos de declaración de cadenas son:

char mensajeLcd[] = “El Lenguaje C”;

define una cadena de 14 caracteres continuos que tienen de forma secuencial los siguientes datos:



Las cadenas o string están formadas por constantes y exige tener un byte adicional para el fin de la cadena.

El C en su liberaría **string.h** contiene gran cantidad de funciones para manipular las cadenas, todas con sus prototipos en el archivo de cabecera.



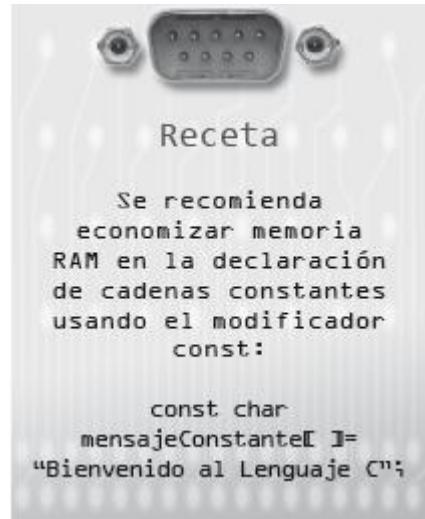
```
mensajeLcd[0] → 'E'  
mensajeLcd[1] → 'T'  
mensajeLcd[2] → ''  
mensajeLcd[3] → 'L'  
mensajeLcd[4] → 'e'  
mensajeLcd[5] → 'n'  
mensajeLcd[6] → 'g'  
mensajeLcd[7] → 'u'  
mensajeLcd[8] → 'a'  
mensajeLcd[9] → 'j'  
mensajeLcd[10] → 'e'  
mensajeLcd[11] → ''  
mensajeLcd[12] → 'C'  
mensajeLcd[13] → 0x00
```

char datosSCI[25]

define una cadena de caracteres no inicializada de 24 caracteres, más 1 para el fin de carácter nulo (0x00).

const char mensaje2[] = “Datos en Flash”;

define una cadena de datos constante que se almacena en memoria de programa o Flash.



6.5.3 Arreglos multidimensionales

De forma similar a los arreglos unidimensionales, la declaración de los arreglos multidimensionales o matrices, se hace de la siguiente forma:

tipo nombre_Matriz[m][n]...[z];

Donde:

tipo: define el tipo de variable de los elementos de la matriz.

nombre_Matriz: define el nombre del array multidimensional o matriz.

m: tamaño de la dimensión m.

n: tamaño de la dimensión n.

...

z: tamaño de la dimensión z.

El acceso a un componente en particular de la matriz deberá tener todos los índices de la matriz.

Los arreglos de datos de más de 3 dimensiones no se usan a menudo por la cantidad de memoria y procesamiento requerido.

La memoria necesaria en memoria esta dado por la siguiente expresión:

nroBytes = sizeof(tipo)*m*n*...*z

El caso más típico de arreglos de datos multidimensionales es el de 2 dimensiones, que define una matriz plana, declarada de la siguiente forma:

tipo matrizXY[m][n];

En la memoria del microcontrolador los datos están de forma secuencial así:

matrizXY[0][0]	matrizXY[0][1]	matrizXY[0][2] matrizXY[0][n]
matrizXY[1][0]	matrizXY[1][1]	matrizXY[1][2] matrizXY[1][n]
matrizXY[2][0]	matrizXY[2][1]	matrizXY[2][2] matrizXY[2][n]
....			
matrizXY[m][1]	matrizXY[m][2]	matrizXY[m][3] matrizXY[m][n]

6.5.4 Estructuras “struct” y uniones “union”

En C, una estructura es un conjunto de variables agrupadas bajo un mismo nombre, proporcionando un medio para mantener ligada mediante un solo ente una información relacionada.

Las variables que conforman la estructura son denominados *campos de la estructura*, cada campo puede ser cualquier tipo de datos de C, con sus respectivos modificadores, un arreglo de datos o una matriz.

La declaración de una estructura en C se realiza de la siguiente forma general:

```
struct Nombre_Estructura { tipo1  
campo1; tipo2 campo2; tipo3  
campo3; .... tipoN campoN; };
```

Donde:

tipoi: es el tipo de datos del campoi. **campoi**: nombre del campo de la estructura.

Y a su vez, la declaración de una localidad de este tipo de estructura:

```
struct Nombre_Estructura nombre_str;
```

Esto reserva espacio en memoria para una estructura de tipo Nombre_Estructura llamada nombre_str.

Para acceder a un campo específico de la estructura a partir del **nombre de la estructura nombre_str**, se hace mediante el operador punto (.) de la siguiente forma general:

```
dato = nombre_str.campo1;
```

El cual entrega el dato contenido en el campoi de la estructura. Y de forma general se puede obtener el apuntador a la estructura así:

```
estructPtr = &nombre_str;
```

Para acceder a un campo específico de la estructura a partir del apuntador a una estructura estructPtr, se hace mediante la secuencia -> de la siguiente forma:

```
dato = estructPtr ->campo1;
```

Como introducción al siguiente ejemplo, y para ilustrar la declaración de una estructura en C, considérese como campos los que describen la información de un alumno de un curso de C, que contiene los siguientes campos:

campo1 : Nombre campo2: Fecha de nacimiento campo3: Edad campo4: Sexo campo5: Estado Civil
campo6: Nota obtenida

Para el manejo de este tipo de estructura se declara Nombre_Estructura así:

```
#define LNG_MAX_NOMBRE 32 // longitud máxima del nombre del alumno  
struct Alumno{
```

```

char nombre[LNG_MAX_NOMBRE];
char birthday[9];           //formato DD/MM/AA
unsigned char edad;          // la edad es siempre positiva
char sexo;                  //sera 'F' (femenino) o 'M' (masculino)
char estadoCivil;           //sera 'S' (soltero), 'C' (casado), 'O' (otro)
unsigned char nota;          //de 0 (mínima) a 5 (máxima)
};


```

La declaración en memoria de un grupo de datos de tipo de esta estructura será:

```
#define NRO_ALUMNOS 6 //número de alumnos del curso C struct Alumno
alumnoCursoC[NRO_ALUMNOS];
```

Que define **6** estructuras en memoria de tipo **Alumno**.

EJEMPLO No. 22

Declaración y manejo de estructuras

Objetivo:

Escribir una rutina en C que reciba el apuntador a una estructura,y retorne el promedio de edad de los alumnos de un curso de la estructura apuntada.

Encender la salida OUT_1 si el curso tiene más alumnos que alumnas,y encender la salida OUT_2 si el curso cuenta con más alumnas que alumnos, encender ambos si existe igual cantidad de alumnos que alumnas.

Solución:

```

//***** Ejemplo 22 *****
//
// Declaración y Manejo de Estructuras
//
// Fecha: Feb 4,2009
// Asunto: Promedio de edades.
//
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// para Microcontrolador FreescaleTM AP16.
// Version: 1.0 Por: Gustavo A. Galeano A.//
//*****



#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /*include peripheral declarations*/



#define DisableWatchdog() CONFIG1_COPD = 1

```

```

#define OUT_1_On() PTC_PTC3 = 1; DDRC_DDRC3 = 1
#define OUT_1_Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1

#define OUT_2_On() PTC_PTC2 = 1; DDRC_DDRC2 = 1
#define OUT_2_Off() PTC_PTC2 = 0; DDRC_DDRC2 = 1

#define LNG_MAX_NOMBRE 32 //long max del nombre

struct Alumno{
    char nombre[LNG_MAX_NOMBRE];
    char birthday[9];      //formato DD/MM/AA
    unsigned char edad;    //edad es siempre positiva
    char sexo;             //'F'(fem) o 'M' (masc)
    char estadoCivil;     //'S'(solt), 'C'(cas), 'O'
    unsigned char nota;   //de 0 (min) a 5 (max)
};

#define NRO_ALUMNOS 6

struct Alumno alumnoCursoC[NRO_ALUMNOS]={
    "Gabriel Garcia", "01/12/70", 37, 'M', 'O', 4,
    "Jaime Florez", "12/12/70", 37, 'M', 'C', 3,
    "Carolina Marin", "01/12/70", 37, 'F', 'C', 4,
    "Natalia Arango", "25/12/76", 32, 'F', 'S', 2,
    "Angela Mejia", "30/09/78", 30, 'F', 'S', 3,
    "Pedro Quintero", "21/02/44", 64, 'M', 'C', 5,
};

unsigned char PromedioEdadCurso(struct Alumno *alumnoPtr){
    unsigned char i;
    unsigned long promedio=0;

    for(i=0;i<NRO_ALUMNOS;i++){
        promedio += alumnoPtr[i].edad;
    }
    promedio /= (unsigned long)NRO_ALUMNOS;
    return (unsigned char)promedio;
}

void main(void) {
    unsigned char i,promedioEdad,nroAlumnas=0,nroAlumnos=0;

    DisableWatchdog();
    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    promedioEdad = PromedioEdadCurso(alumnoCursoC);
}

```

```

// cálculo del nro de alumnos y alumnas
for(i=0;i<NRO_ALUMNOS;i++){
if((alumnoCursoC[i].sexo) == 'M'){
    nroAlumnos++;
}else{ // no es 'M'
    nroAlumnas++;
}
}
if(nroAlumnos >= nroAlumnas){
    OUT_1_On();
}
if(nroAlumnas >= nroAlumnos){
    OUT_2_On();
}
for(;;) {} /* loop forever */
/* please make sure that you never leave main */
}

```

Discusión:

Nótese la declaración de la plantilla **struct Alumno**, la cual no ocupa memoria, una vez declaramos la estructura **struct Alumno alumnoCursoC[NRO _ALUMNOS]**, se están declarando 6 estructuras de tipo Alumno, la cual de forma análoga a la declaración de una variable, se puede inicializar.

La rutina **PromedioEdadCurso()** no recibe una estructura, sino un apuntador (2 bytes) a la estructura sobre la cual va a procesar.

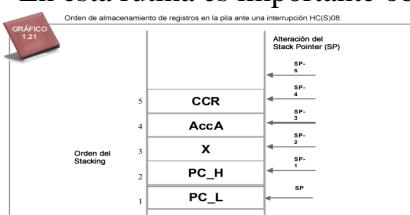
En esta rutina es importante observar además que para obtener el promedio, se necesita

la suma aritmética de los **6** alumnos, dato debe ser almacenado en un tipo **ulong**, para
 evitar desborde de la suma, una vez se tiene este resultado se divide por el número de
 alumnos y se obtiene un ulong menor que 255, y dado que la función retorna un uchar,
 hacemos uso del molde para ajustar el resultado al retorno de la función.

Al final y para resolver la ultima parte del enunciado,

nos ayudamos de un ciclo for que recorre el campo
 “sexo” para determinar el número de alumnos y de
 alumnas.

Observar en el Gráfico 6.2 la forma como esta
 posicionada la estructura en memoria RAM del
 microcontrolador.





Almacenamiento en memoria de los campos de una estructura.

The screenshot shows a debugger interface with three windows:

- Source**: Displays the C source code for a program named `main.c`. It includes a function `promedio` that calculates the average age of students in a course, and a `main` function that initializes a `alumnoCursoC` array and calls `promedio`.
- Data**: Shows the memory layout of the `alumnoCursoC` array. The array has 6 elements, each representing a student (`Alumno`). The fields for each student are:
 - `nombre`: array[32] of unsigned char
 - `edad`: 1 byte unsigned char
 - `sexo`: 1 byte unsigned char
 - `estadocivil`: 1 byte unsigned char
 - `nota`: 4 bytes unsigned char
- Memory**: Displays the raw memory dump for the `alumnoCursoC` array. The memory starts at address `0x100` and contains 6 entries, each 40 bytes long. The entries represent student records with fields: nombre, edad, sexo, estadocivil, and nota.

A yellow arrow points from the `alumno` field in the Data window to the first student entry in the Memory window. Another yellow arrow points from the `edad` field in the Data window to the second byte of the first student entry in the Memory window.

Almacenamiento en memoria de los campos de la primera estructura alumnoCursoC.

Alumno	edad	sexo	estadocivil	nota
Pablos	20	M	S	90
Carla	21	F	S	85
Laura	22	F	S	88
José	23	M	S	92
Patricia	24	F	S	89
Diego	25	M	S	91

Clíc para ver en alta resolución

6.5.5 Estructuras de bits

Las **estructuras de bits** nacen como una necesidad de economizar espacio en memoria, tienen la misma connotación que las estructuras de datos, pero en esta un bit o conjunto de bits conforman un campo de la estructura.



En C el tipo de dato más pequeño es el `char`, el cual puede adoptar 255 valores, por ello se dividen sus 8 bits en campos de bits usadas

de forma separada.



un bit así:

bit en 0 es Masculino, bit en 1 es Femenino, y de forma similar el campo nota, requiere 6 valores posibles: 0,1,2,3,4,5, que puede ser almacenada en 3 bits, que permite incluso 7 valores (uno más de los necesarios).

En sistemas embebidos es común usar bits de una palabra, que identifican banderas que solo toman 2 valores, 0 ó 1, indicando si la bandera está activa o inactiva.

La declaración y acceso de las estructuras de bits está basada en la misma de las estructuras, indicando después del campo seguido por el símbolo “:” el número de bits usados de la palabra así:

```
struct Nombre_Estructura{  
    tipo nombreCampoBits1 : p;  
    tipo nombreCampoBits2 : q;  
    tipo nombreCampoBits3 : r;  
    ....  
} nombre_StrBits ;
```

Donde:

p,q,r: definen el número de bits requeridos respectivamente para **nombreCampoBits1**, **nombreCampoBits2** y **nombreCampoBits3**.

Los campos individuales de la estructura se referencian de la misma forma que se accesa una estructura así:

```
nombre_StrBits.nombreCampoBits1  
nombre_StrBits.nombreCampoBits1  
nombre_StrBits.nombreCampoBits1
```

El caso más típico de estructuras de bits lo conforma el acceso a bits de los registros internos del microcontrolador.

Para ilustrar el uso de este tipo de estructuras, consultar el ejemplo No. 23: Estructuras de bits en C.

EJEMPLO No. 23

Estructuras de bits en C

Objetivo:

Escribir un programa en C que indique en banderas de bits los siguientes eventos:

key_1_was_pressed: en 1 indica si se ha presionado la tecla **INPUT_1**.

key_2_was_pressed: en 1 indica si se ha presionado la tecla **INPUT_2**.

key1_2_were_pressed: en 1 indica que se ha presionado la tecla **INPUT_1** y la **INPUT_2** al mismo tiempo.

key_1_is_mod_7: en 0 indica que el número de presiones de la tecla **INPUT_1** es múltiplo exacto de 7.

flag_Z_status: indica el estado de la bandera Z del CCR una vez que es llamada la función **CheckZ()**.

La salida **OUT_1** debe activarse si la bandera **key_1_is_mod_7** esta activa, indicando que el número de presiones es múltiplo de 7.

Solución:

```
***** Ejemplo 23 *****
// Estructuras de Bits en C
// Fecha: Marzo 31,2009
// Asunto: Manejo de banderas de bits.
// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)
// para Microcontrolador PIC 16F877A.
// Versión: 1.0 Por: Gustavo A. Galeano A.
*****
#include "structBits.h"
#define OUT_1_On() output_high(PIN_B4)
#define OUT_1_Off() output_low(PIN_B4)
#define INPUT_1() !input(PIN_B0)
#define INPUT_2() !input(PIN_B1)
//Función de inicialización del microcontrolador
void Mcu_Init(void){
    Mcu_Init_PIC(); /*cuerpo de la función en Ejemplo 4*/
}
```

```

struct Flags_Teclas{
    char flag1: 1;
    char flag2: 1;
    char flag3: 1;
    char cnt1: 3;
    char flagZ: 1;
    char nkey1: 1;
};

struct Flags_Teclas flags_keys;
#define key_1_was_pressed flags_keys.flag1
#define key_2_was_pressed flags_keys.flag2
#define key1_2_were_pressed flags_keys.flag3
#define KEY_1_IS_MOD_7 flags_keys.cnt1
#define FLAG_Z_STATUS flags_keys.flagZ
#define FLAG_N_KEY_1 flags_keys.nkey1
void FlagZ_Update(void){ //actualiza el estado de la bandera Z del CCR
char estadoZ=0;
#asm
    movf 0x03,w
    movwf estadoZ
#endasm
    estadoZ &= 0b0000100;
    if(estadoZ){ FLAG_Z_STATUS = 1;}else{ FLAG_Z_STATUS = 0; }
}
void Flags_Init(void){ // Inicializa la estructura flags_keys
    key_1_was_pressed = 0;
    key_2_was_pressed = 0;
    key1_2_were_pressed = 0;
    KEY_1_IS_MOD_7 = 0;
    FlagZ_Update();
}
void main(){//Funcion principal
    Mcu_Init();
    // TODO: USER CODE!!
    Flags_Init();
    for(;;) {
        if(INPUT_1() && !KEY_1_WAS_PRESED) KEY_1_WAS_PRESED = 1;
        if(INPUT_2() && !KEY_2_WAS_PRESED) KEY_2_WAS_PRESED = 1;
        if(INPUT_1() && INPUT_2 ()) KEY1_2_WERE_PRESED = 1;
        if(INPUT_1() && !FLAG_N_KEY_1){
            FLAG_N_KEY_1 = 1;
            KEY_1_IS_MOD_7++;
            if(!KEY_1_IS_MOD_7){
                OUT_1_On();
            }else{
                OUT_1_Off();
            }
        }
    }
}

```

```

if(!INPUT_1()) FLAG_N_KEY_1 = 0;
} /* loop forever */
}

```

Discucion:

Sin duda se pudiera solucionar el enunciado usando un byte y realizando macros (#define's), con ANDs sobre cada uno de los bits; sin embargo, se perdería claridad y para la implementación del contador cnt1, se tendría que acudir a otro byte.

En esta solución se acude a la estructura de bits que permite usar tantos bits como sea necesario del byte de la estructura flags_keys.

Al realizar la declaración de la estructura se definen los nombres de cada uno de los campos y el número de bits que usa cada bandera, en el caso de flag1,flag2, flagZ,nkey1 solo se requiere un bit (0 para un estado, 1 para otro estado); sin

embargo, para el campo cnt1 se requieren 3 bits, debido a que se requiere contar de 0 (000) a 7 (111).

Es importante anotar que al definir la estructura:

```

struct Flags_Teclas{
char flag1: 1;
char flag2: 1;
char flag3: 1;
char cnt1: 3;
char flagZ: 1;
char nkey1: 1;
};

```

Solo se usa 1 byte y sobre él se posicionan los bits necesarios para cada uno de los campos.

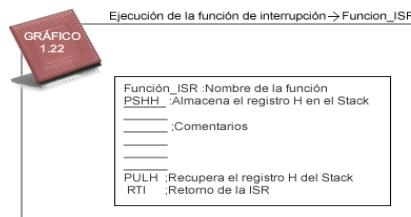
El funcionamiento con la declaración:

```

struct Flags_Teclas{
char flag1;
char flag2;
char flag3;
char cnt1;
char flagZ;
char nkey1;
};

```

Sería exactamente el mismo y de hecho funcionaría de forma equivalente, pero en lugar de 1 byte de RAM ocuparía 6 bytes.



El uso de este tipo de estructuras de bits en general es muy usado en sistemas embebidos para indicar eventos, suceso de timeouts, validación de alguna condición que solo requiera un bit.

6.6 APUNTADORES A FUNCIONES

En el lenguaje C, las funciones son una sección de código que tiene una dirección de inicio y una dirección de fin o dirección de retorno.

El **apuntador a una función** es una variable que tiene la dirección de inicio a la función, la que es útil en administración de tareas como lo son los sistemas operativos de tiempo real o estructuras de programación de funcionamiento dinámico.

Para una función definida con prototipo:

```
retorno Nombre_Funcion(arg1, arg2, ..., argN);
```

El apuntador a la función es:

```
Nombre_Funcion
```

Lo más útil y claro, para declarar los apuntadores a funciones es hacerlo mediante una definición de tipo **typedef**:

```
typedef retorno (*rFuncPtrA)(args);
```

Donde:

retorno: es el tipo de dato returnedo por la función.

args: es la lista de los tipos de argumentos que recibe la función.

rFuncPtrA: es el nombre del tipo apuntador.

Una localidad de memoria se declara:

```
rFuncPtrA FuncionPtr;
```

EJEMPLO No.24

Apuntadores a funciones

Objetivo:

Realizar un ciclo de llamado a funciones secuenciales cuyos apuntadores se encuentran en una estructura que contiene la dirección de inicio de cada una de las funciones que se quieren procesar.

Ejecutar solo las funciones que tienen dirección válida (diferente de NULL).

Realizar la prueba con diversas funciones de encendido, apagado de los leds OUT_1,OUT_1, BUZZER, INPUT_1 y de INPUT_2.

Solución:

```
***** Ejemplo 24 *****
//
// Apunntadores a Funciones
//
// Fecha: Feb 4,2009
// Asunto: Llamado de funciones
// de forma secuencial.
//
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// para Microcontrolador Freescale AP16.
// Version: 1.0 Por: Gustavo A. Galeano A.
//
*****



#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /*include peripheral declarations*/
#define DisableWatchdog() CONFIG1_COPD = 1
#define OUT_1_On() PTC_PTC3 = 1; DDRC_DDRC3 = 1
#define OUT_1_Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1
#define OUT_2_On() PTC_PTC2 = 1; DDRC_DDRC2 = 1
#define OUT_2_Off() PTC_PTC2 = 0; DDRC_DDRC2 = 1
#define INPUT_1() !PTD_PTD1
#define INPUT_2() !PTD_PTD2
#define BUZ_On() PTD_PTD0 = 1; DDRD_DDRD0 = 1
#define BUZ_Off() PTD_PTD0 = 0; DDRD_DDRD0 = 1

void Tarea_1(void);
void Delay(void);
void Tarea_2(void);
void Funcion_3(void);
void Funcion_Cnt(void);

typedef void (*vFuncPtrV)(void);
```

```

#define NRO_FXS 6

vFuncPtrV arrayFxs[NRO_FXS]={
    Tarea_1,
    Delay,
    Tarea_2,
    NUL,
    Funcion_3,
    Funcion_Cnt
};

//Funcion de eliminado de la tarea del arreglo arrayFxs
void ClearFx(vFuncPtrV funcionPtr){
uchar i;
    for(i=0;i<NRO_FXS;i++){
        if(arrayFxs[i] == funcionPtr) arrayFxs[i]= NULL;
    }
}

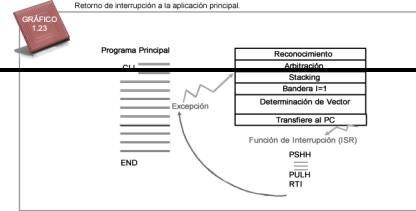
//Funcion Delay()--> retardo
void Delay(void){
unsigned long i;
    for(i=0;i<2000;i++);
}

//Fx Tarea_1() Enciende OUT_1 al presionar la tecla 1 y
// apaga OUT_1 si la tecla 1 no está presionada
void Tarea_1(void){
    if(INPUT_1()){
        OUT_1_On();
    }else{
        OUT_1_Off();
    }
}

//Funcion Tarea_2() enciende y apaga OUT_2 20 veces
void Tarea_2(void){
static uchar nroBlinks;

    if(nroBlinks < 20){
        OUT_2_On();
        Delay();
        OUT_2_Off();
        Delay();
        nroBlinks++;
    }else{
        ClearFx(Tarea_2);
    }
}

```



```
//Fx Funcion_3() enciende Buzzer si tecla 2 es presionada
void Funcion_3(void){
    if(INPUT_2()){
        BUZ_On();
    }else{
        BUZ_Off();
    }
}
```

```
//Fx Funcion_Cnt() incrementa el valor de variable contador
void Funcion_Cnt(void){
    static ulong contador;
    contador++;
}
```

```
void main(void) {
uchar i;

DisableWatchdog();
EnableInterrupts; /* enable interrupts */
/* include your code here */
i=0;
for(;;) {
    if(arrayFxs[i] != NULL) arrayFxs[i]();
    i = (i+1)%NRO_FXS;
} /* loop forever */
/* please make sure that you never leave main */
}
```

Discusión:

Como se puede observar el `typedef` permite crear un tipo de datos propio que en este caso es un apuntador, y específicamente a una función tipo `void Fx(void)`.

Se declara una estructura de datos cuyo contenido son los apuntadores a función, y los inicializamos apuntando a diferentes funciones: `Tarea_1`, `Delay`, `Tarea_2`, `Funcion_3` y `Funcion_Cnt`.

El segundo ciclo `for` del `main()` verifica si el apuntador no es `NULL` y de ser así toma el apuntador y realiza un llamado a subrutina, el cual es equivalente en ensamblador a un `JSR FuncionPtr`.

Dado que la `Tarea_2` deberá ejecutarse solo 20 veces, una vez se cumplen, se invoca la función `ClearFx()` que recibe un apuntador a tipo `vFuncPtrV` y elimina el apuntador del array `arrayFxs`.

6.7 MANEJO DE INTERRUPCIONES EN SISTEMAS EMBEBIDOS DESDE C



Las interrupciones son procedimientos que tienen un tratamiento especial, diferente a una función de C normal.



Todo sistema embebido, por sencillo que sea, maneja interrupciones de diferente tipo, los compiladores usan diferente sintaxis de tal forma que no hay una regla general que los cubra a todos; sin embargo, el concepto de función de interrupción ISR (*Interrupt Service Routine*) si es general, y se refiere a una función que el software no invoca, sino que es el procesador quien al detectar una interrupción (o excepción), cambia el contexto hacia esta función.

Esta ejecutará su código completo y retornará con una instrucción diferente a la de retorno de función normal, debido a que tiene que recuperar del stack no solo la dirección de retorno sino el contenido de cada uno de los registros del modelo de programación.

En cada interrupción que se defina se deberán tener en cuenta sin excepción los siguientes 6 puntos:

1. Las funciones de interrupción o ISR, son diferenciadas de las funciones o subrutinas normales anteponiéndole la palabra reservada **interrupt**, o Directiva TRAP-PROC la cual le indica al compilador que la función en cuestión es una ISR y lo prepara para generar un retorno de interrupción (RTI) en lugar de un retorno de subrutina normal (RTS). Además, el compilador prepara el código generado para conservar el modelo de programación que no es guardado en el stack por el cambio de contexto de la CPU (para el caso del HC(S)08 generar las instrucciones PSHH y PULH al comienzo y fin de la función de interrupción para guardar y recuperar el registro H respectivamente al inicio y fin de la función de interrupción).
2. Tal como se vio en el Capítulo 1 (numeral 1.5.2), una vez se genera una interrupción el procesador tomará una y solo una dirección de la tabla de vectores de interrupción, en la cual está la dirección de la función a la que deberá ir una vez se cambie el contexto (PC = Vector de Interrupción), este vector se genera una vez se realiza la definición de la función mediante un número **NroVector** que sigue a la palabra reservada **interrupt**, dicho número se cuenta a partir del vector de RESET, el cual es CERO, hasta llegar al vector al que corresponde la interrupción que se quiere invocar. A continuación se verá la tabla de interrupciones para el microcontrolador AP16A:

TABLA
6.1

Vectores de interrupción del AP16A.

Dirección Vector	Interrupción	Prioridad	L
0xFFD0 – 0xFFD1	Reserved – No usado	23	
0xFFD2 – 0xFFD3	Timebase – Base de Tiempo	22	
0xFFD4 – 0xFFD5	Ir SCI Tx – Transmisión Ir SCI	21	
0xFFD6 – 0xFFD7	Ir SCI Rx – Recepción Ir SCI	20	
0xFFD8 – 0xFFD9	Ir SCI Err – Error en Ir SCI	19	
0xFFDA – 0xFFDB	SPI Tx – Transmisión SPI	18	
0xFFDC – 0xFFDD	SPI Rx – Recepción SPI	17	
0xFFDE – 0xFFDF	ADC - Conversión Completa	16	
0xFFE0 – 0xFFE1	Keyboard – Pines Puerto D	15	
0xFFE2 – 0xFFE3	SCI Tx – Transmisión de SCI	14	
0xFFE4 – 0xFFE5	SCI Rx – Recepción de SCI	13	
0xFFE6 – 0xFFE7	SCI Err – Error de SCI	12	
0xFFE8 – 0xFFE9	MMIIC – Serial I2C	11	
0xFFEA – 0xFFEB	TIM2 Ov – Timer2 Sobre flujo	10	
0xFFEC – 0xFFED	TIM2 Ch 1 – Timer2 Canal 1	9	
0xFFEE – 0xFFEF	TIM2 Ch 0 – Timer2 Canal 0	8	
0xFFFF0 – 0xFFFF1	TIM1 Ov – Timer1 Sobre flujo	7	
0xFFFF2 – 0xFFFF3	TIM1 Ch 1 – Timer1 Canal 1	6	
0xFFFF4 – 0xFFFF5	TIM1 Ch 0 – Timer1 Canal 0	5	

Vectores de Interrupción del AP16A.

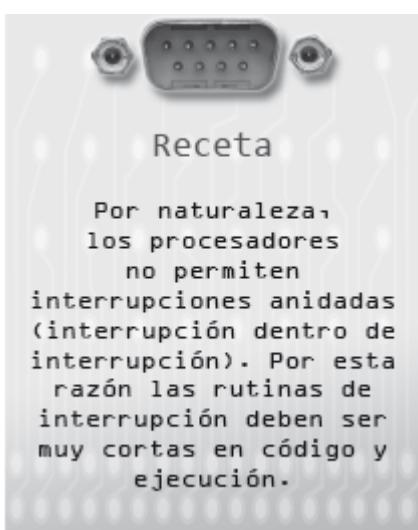
3. Por el hecho de no ser invocadas desde el programa, la ISR no recibe argumentos (void), y no retorna resultado (**void**). Se recomienda acompañar el nombre de la función con la terminación ISR, para indicar explícitamente que se trata de una función de interrupción.

El prototipo de una función de interrupción se verá de la siguiente forma general:

```
interrupt NroVector void Funcion_ISR(void);
```



El código de programa de una interrupción suele ser por lo general corto para evitar demoras en el retorno de la ISR (Interrupt Service Routine).



4. En términos generales el código de programa de la interrupción deberá ser un código muy corto, lo más resumido posible a fin de evitar demoras en el retorno de la ISR, debido a que como se recordará, el cambio de contexto desde el programa principal hacia la interrupción deshabilitan todas las interrupciones para evitar anidación (bandera I en el CCR en 1). Por consiguiente, cualquier interrupción que ocurra durante la ejecución del código de la interrupción, quedará pendiente y solo será atendida al retorno de ésta, inclusive si la interrupción es de mayor prioridad a la que se está llevando a cabo. Existe una técnica que permite el manejo anidado de interrupciones, la cual consiste en habilitar al inicio de la ISR las interrupciones mediante la invocación del macro ENABLE_INTERRUPTS; (o asm ("CLI")), de esta forma, si una nueva interrupción sucede suspenderá la presente ISR e iniciará cambio de contexto hacia la nueva ISR. Sin embargo, no es recomendable en los sistemas embebidos porque genera inestabilidad sobre el nivel de uso del stack.

Recuérdese que el stack es una zona de RAM que se comparte con los datos de uso del programa, si el stack pierde control puede suceder que afecte los datos de almacenamiento de variables del programa mismo.

5. En cada ISR debe borrarse la bandera correspondiente a la interrupción que genera el llamado a la ISR, a fin de no ocasionar el llamado consecutivo a la misma ISR ante la misma interrupción. El borrado de cada bandera deberá consultarse en el manual del procesador usado, debido a que cada bandera puede tener una forma exclusiva y precisa de borrado.
6. Recordar que las variables en RAM que se usen dentro de una ISR, deberán tener el modificador "**volatile**", que asegura que su código, redundante u obvio, no será optimizado.

Finalmente, la implementación de una ISR se verá de la siguiente forma general:

```
interrupt NroVector void Funcion_ISR(void){  
    ... código de programa de la interrupción  
    ... código de programa borrado de bandera de interrupción  
}
```

6.7.1 Configuración de interrupciones en Codewarrior ®

El manejo de la ISR en el ambiente del Codewarrior® tiene varias formas dependiendo como el grupo de programadores lo sienta más claro y lo pueda modificar de forma más segura y sencilla.

El objetivo final en todos los casos consiste en llenar la tabla de vectores de interrupción con las direcciones de las funciones ISR apropiadas para cada llamado.

Opción 1: es la forma tradicional en la cual se determina NroVector a partir de la tabla de interrupción y se acompaña en la implementación de la interrupción.

Consideremos que se usará la interrupción de Timebase – Base de Tiempo, para cuyo caso particular NroVector será 22, luego la implementación sería:

```
interrupt 22 void TimeBase_ISR(void){  
    ... código de programa de la interrupción  
    ... código de programa borrado de bandera TimeBase  
}
```



Para manejar las interrupciones en Codewarrior® los programadores deben llenar de forma adecuada la tabla de vectores de interrupción con las direcciones de las funciones ISR apropiadas para cada llamado.

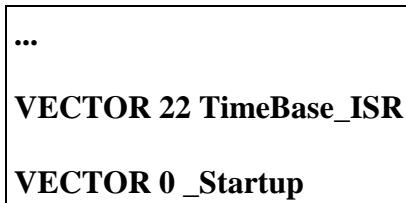


Opción 2: modificar la dirección de la ISR de las interrupciones usadas en el archivo **ProjectName.prm**, propio de cada proyecto.

En la carpeta **Linker Files** creada para cada proyecto, se abre el archivo con extensión.prm Project→Name.prm, se define la dirección a la cual el procesador direccionará cada interrupción.

El vector 0 (RESET), apunta allí a la dirección de la función de inicio _Startup (o arranque), que luego invoca a la función main().

Para direccionar un nuevo vector, por ejemplo el **TimeBase (nroVector 22)** hacia la función **TimeBase_ISR**, se modifica el archivo **ProjectName . prm** así:



Y la función de interrupción ya no requiere la definición de la Opción 1, quedando así:

```
interrupt void TimeBase_ISR(void){  
    ... código de programa de la interrupción  
    ... código de programa borrado de bandera TimeBase  
}
```

Opción 3: esta opción consiste en crear un archivo en C, normalmente llamado **vectors.c** que contiene una tabla de apunadores a las diferentes funciones de interrupción.

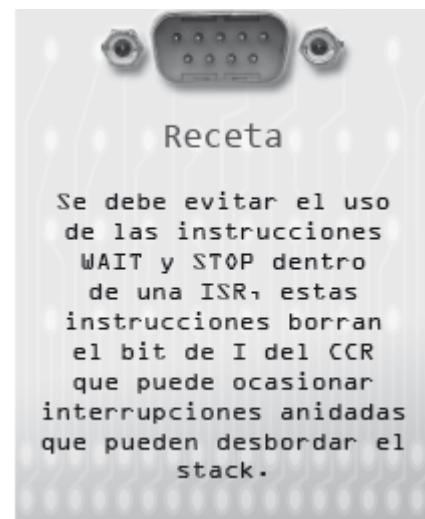
Cada que una interrupción nueva es creada, se accede a vectors.c y se modifica el vector de la interrupción correspondiente.

Así, si se quiere usar la interrupción de base de tiempo TimeBase, se declara así:

```
interrupt void TimeBase_ISR(void){  
    ... código de programa de la interrupción  
    ... código de programa borrado de bandera TimeBase  
}
```

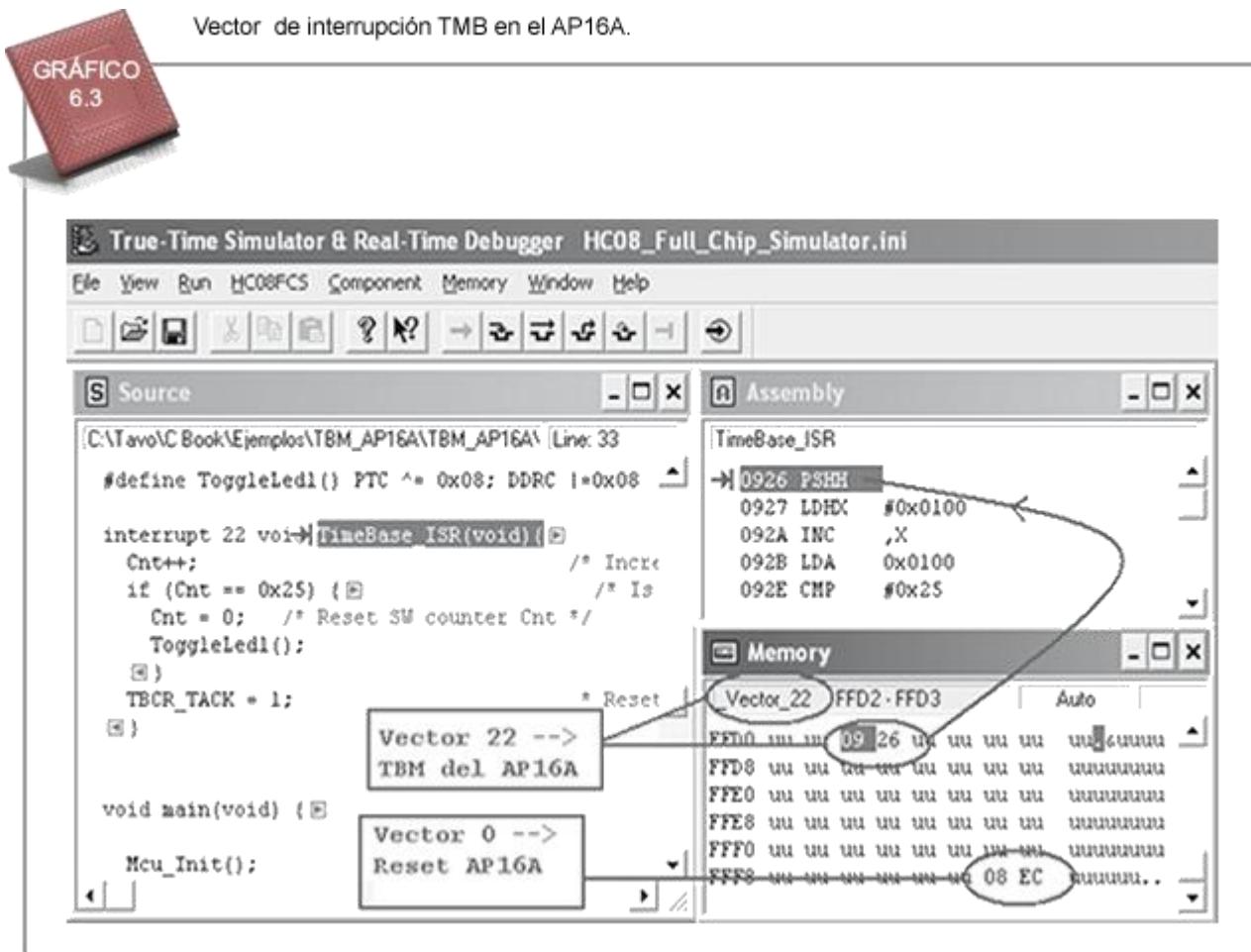
Y se modifica el archivo vectors.c de la siguiente forma:

```
void (* const _vect[])() @0xFFD2 = { // Inicio Tabla de Interrupciones AP16A
    TimeBase_ISR, //Vector 22 → Time Base apunta a TimeBase_ISR()
    Cpu_Interrupt, //Vector 21 → IRSCITransmit
    Cpu_Interrupt, //Vector 20 → IRSCIReceive
    Cpu_Interrupt, //Vector 19 → IRSCIError
    Cpu_Interrupt, //Vector 18 → SPITransmit
    Cpu_Interrupt, //Vector 17 → SPIReceive
    Cpu_Interrupt, //Vector 16 → ADC
    Cpu_Interrupt, //Vector 15 → KBD
    Cpu_Interrupt, //Vector 14 → SCITransmit
    Cpu_Interrupt, //Vector 13 → SCIReceive
    Cpu_Interrupt, //Vector 12 → SCIError
    Cpu_Interrupt, //Vector 11 → MMIIC
    Cpu_Interrupt, //Vector 10 → TIM2Ovr
    Cpu_Interrupt, //Vector 9 → TIM2CH1
    Cpu_Interrupt, //Vector 8 → TIM2CH0
    Cpu_Interrupt, //Vector 7 → TIM1Ovr
    Cpu_Interrupt, //Vector 6 → TIM1CH1
    Cpu_Interrupt, //Vector 5 → TIM1CH0
    Cpu_Interrupt, //Vector 4 → PLL
    Cpu_Interrupt, //Vector 3 → IRQ2
    Cpu_Interrupt, //Vector 2 → IRQ1
    Cpu_Interrupt, //Vector 1 → SWI Software _
    Startup //Vector 0 → Reset
};
```



Aunque todas las opciones son válidas, se recomienda la Opción 1 porque permite mejor modularidad de los diferentes archivos que conforman un proyecto.

Vector de interrupción TBM en el AP16A.



EJEMPLO No. 25

Base de tiempo real Gráfico

Objetivo:

Realizar un código en C usando el módulo TBM del AP16 que permita tener una base de tiempo real de 1 segundo, invertir el estado de la salida OUT-1 (LED) cada segundo para

validar el funcionamiento del código.

El código del programa principal, solo deberá inicializar los módulos y quedar en el loop:

```
for(;;){  
}
```

Sin ningún código adicional.

Crear una variable unsigned long que contiene el número de segundos actuales desde que el programa fue iniciado.

Solución:

La frecuencia del oscilador externo es de 9.8304MHz, con lo que la frecuencia interna es:

$$9.8304\text{MHz} / 4 \diamond 2.4576\text{MHz}, \text{OSCCLK } 0.41\text{uSeg}$$

Siendo el timer del TBM de 16 bits, ocurrirá un sobreflujo cada $0.41\text{uSeg} * 65536 = 26.6667\text{mSeg}$

Para contabilizar 1 segundo, se deben contabilizar $1000/26.67 = 37.5$ **interrupciones**.

```
// Código en C que configura una interrupción TBM para contabilizar 1 Segundo  
//***** Ejemplo 25 *****
```

```
//  
// Base de Tiempo Real  
//  
// Fecha: Feb 4,2009  
// Asunto: Base de tiempo de 1 segundo,  
// Usando el modulos TBM del AP16A.  
//  
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)  
// para Microcontrolador Freescale AP16.  
// Version: 1.0 Por: Gustavo A. Galeano A.  
//  
//*****
```

```
#include <hdef.h> /* for EnableInterrupts macro */  
#include "derivative.h" /* include peripheral declarations */
```

```
static byte Cnt;
```

```
/* MOR: OSCSEL1=1,OSCSEL0=1,??=1,??=1,??=1,??=1,??=1 */  
const unsigned char MOR_INIT @0x0000FFCF = 0xFF;
```

```

void Mcu_Init(void){
    /* CONFIG1: COPRS=0,LVISTOP=0,LVIRSTD=0,LVIPWRD=0,LVIREGD=0,SSREC=0,
STOP=1, CO
PD=1 */
    CONFIG1 = 0x03;
    /*
STOP_ICLKDIS=0,STOP_RCLKEN=0,STOP_XCLKEN=0,OSCCLK1=1,OSCCLK0 =0,??
=0,?=0,SCIBDSRC=0 */
    CONFIG2 = 0x10;
}

static void SetPV(byte Val){
    TBCR_TBON = 0;      /* Disable and reset device */
    TBCR_TBR = Val;    /* Store given value to the prescaler */
    TBCR_TBON = 1;
}

void BaseTiempo_Init(void){
    TBCR = 0x0C; /* Initialize timebase */
    Cnt = 0;
    SetPV((byte)0x00); /* Set prescaler register according to the selected high
speed CPU mode */
    /* TBCR: TBON=1 */
    TBCR |= 0x02; /* Habilita Timer*/
}
#define ToggleLed1() PTC ^= 0x08; DDRC |=0x08

interrupt 22 void TimeBase_ISR(void){
    Cnt++; /* Increment SW counter Cnt */
    if (Cnt == 0x25) { /* Is it now the period time? */
        Cnt = 0; /* Reset SW counter Cnt */
        ToggleLed1();
    }
    TBCR_TACK = 1; /* Reset interrupt request flag */
}

void main(void) {
    Mcu_Init();
    BaseTiempo_Init();
    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    for(;;) {
    } /* loop forever */
}

```



Discusión:

Para la implementación del ejemplo se configuró el funcionamiento continuo de la interrupción de base de tiempo TBM, calculando inicialmente la periodicidad de la interrupción a partir del oscilador externo.

El cálculo arroja que en 37.5 interrupciones se cumple un segundo, el programa se puede refinar para que se contabilice un segundo exacto realizando el conteo de 38 interrupciones en un segundo y en el próximo 37 ciclos.

Se puede además observar que el `for(;;){}` (loop continuo) del `main()`, muestra que el programa principal no realiza ninguna acción, y toda la actividad se realiza dentro de la interrupción cuando se ha cumplido un segundo.

Dentro de la interrupción por su parte se realiza el borrado de la bandera de interrupción TACK, lo cual se hace escribiendo un “1” en su valor, de esta forma se reconoce la interrupción actual.

El ciclo `for(;;){}` es interrumpido cada 26.67mSeg solo para incrementar una variable que no toma mayor tiempo, con esto el tiempo de ocio del microcontrolador es de más de un 99% disponible para realizar otras labores.

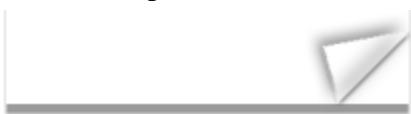
6.8 CONVENCIONES ÚTILES DE PROGRAMACIÓN EMBEBIDA EN C



Las convenciones son protocolos adoptados por los desarrolladores de un proyecto para evitar la pérdida de tiempo que representa el que cada cual trabaje como más le parezca.

Cuando todos los miembros del equipo adoptan el mismo set de reglas es fácil que un programador modifique el código desarrollado por uno

de sus compañeros.



Se consiguen gran cantidad de beneficios en productividad cuando en un proyecto que está siendo trabajado en equipo o una corporación, adopta una convención consistente en la programación.

Esta sección muestra algunas técnicas útiles para manejar los proyectos en C, como declarar las variables, las constantes, las funciones y demás elementos del proyecto.

Puede generar un poco de controversia porque cada uno de los programadores tiende a programar como mejor le parece, y eso no significa que se esté haciendo bien o mal.

Sin embargo, es importante que todos los miembros del equipo adopten un set sencillo de reglas que sean seguidas de forma estándar por todos los participantes, de tal forma que el código desarrollado por uno de los integrantes, pueda ser fácilmente modificable por otro o por sí mismo, tiempo después cuando el producto requiere cambios, actualizaciones o customizaciones que nunca faltarán.

Se debe tener en cuenta que el trabajo como programador no termina cuando el código trabaja, sino cuando se escribe un código que además de funcionar es de fácil mantenimiento.

Una de las formas más sencillas para lograr este objetivo consiste en adoptar una forma limpia y consistente de codificar. El estilo que se decida no importará, sí y solo sí un programador particular y los demás integrantes de su organización también lo adoptan de una forma común.

6.8.1 Sobre la Distribución del Código Fuente

El código fuente deberá ser agrupado por módulos, cada módulo deberá tener como mínimo dos (2) archivos, uno extensión .C y otro de extensión .H. Los archivos de cabecera (.H) nunca deben contener código, excepto los macros.

Un archivo de código fuente debe contener las siguientes secciones en este orden:

Cabecera del archivo:

Título:

Nombre Archivo:

Fecha : Mayo 2, 2008

Version: , Revisión por autor

Asunto: Historia de revisiones #includes #defines y macros variables Prototipos de funciones Funciones públicas Funciones privadas (static)

La cabecera del archivo es un bloque de comentario que contiene el nombre de la compañía, la dirección, una nota de derechos de copia (copyright), el nombre del archivo, el nombre del autor y una pequeña descripción del módulo.

La historia de revisión es un bloque de comentario que describe los cambios que se han realizado en el pasado, algún software de control de versión pueden llenar este campo de forma automática.

Los **#defines** y los macros pueden estar localizados en tres lugares diferentes, en primer lugar si los **#define** y los macros solo son usados dentro del módulo estos estarán localizados en el archivo del módulo (.C), así por ejemplo, para los diferentes estados de la máquina, no tienen sentido en extender la visibilidad de algo que solo se va a usar de forma local; segundo, si los #defines y macros se van a usar en diferentes módulos, éstos deberán ir en el archivo de cabecera (.H) con el objetivo de darles visibilidad global a varios módulos; por último, si el #define significa algo específico de la aplicación, deberá ir en el archivo APP.H, así por ejemplo, si el #define indica alguna compilación condicional o habilita o deshabilita alguna característica final al producto.

En la sección **#includes** van incluidos todos los archivos de cabecera que el módulo requiere para compilar. Una forma sencilla y práctica para los includes, es crear un archivo maestro llamado **GLOBAL.H o INCLUDES.H**, que incluye a todos los archivos del proyecto, de tal forma que los nuevos archivos solo requieren incluir a este único archivo para compilar, esto previene de recordar que archivo de cabecera incluye qué. El único inconveniente que tiene esta práctica es que la compilación de cada archivo se puede hacer un poco más lenta.

6.8.2 Sobre los nombres de funciones



La creación de un archivo maestro llamado **GLOBAL.H o INCLUDES.H** permite manejar en un solo lugar todos los archivos de cabecera que demande el proyecto.



Los nombres de las funciones empiezan con letra mayúscula y deberían tener un prefijo del nombre del módulo, así como un modificador static para encapsular su acceso desde el exterior en el caso de aquellas que únicamente actúan en

su propio módulo.

Las variables comienzan por una letra minúscula y pueden constar de combinaciones alfanuméricas; las mayúsculas se usan para separar palabras dentro de la variable.

Cuando se crean nombres de funciones (y de forma similar a las variables), es usual usar acrónimos, como OS, ISR, Init, Run, Flush, y mnemónicos, como cir, cmp etc.

Esto le permite al programador ser descriptivo requiriendo pocos caracteres.

Los nombres de las funciones deberán iniciarse por una letra mayúscula, seguida de combinaciones alfanuméricas mayúsculas y minúsculas, separadas por el carácter ‘_’ underscore. Cada función debe tener un prefijo del nombre del módulo y es útil un ‘_’ underscore a continuación. Así por ejemplo, un módulo de comunicación serial puede tener las siguientes funciones para inicializar, procesar, parar y destruir su funcionamiento:

```
char Serial_Init(void); void Serial_Process(void); char  
Serial_Hold(void); char Serial_Flush(void);
```

Es una buena práctica también que funciones únicamente usadas dentro de un módulo tengan el modificador static para encapsular y esconder su acceso desde el exterior.

Las variables locales dentro de estas funciones deberán declararse en una sola línea para permitir realizar el comentario en la respectiva línea, y empezar el código de la función dos (2) espacios abajo después de la última variable local, esto permite claridad y delineación entre las variables y el código ejecutable.

La implementación de una función como el módulo sugerido anteriormente se vería así:

```
char Serial_Init(void){  
    unsigned char i; //contador  
    bool retorno;  
    unsigned int j;  
    SCI1_BR = REG_VAL;  
    retorno = SCI1SR;  
    return retorno;  
}
```

6.8.3 Sobre las variables y apuntadores



Las constantes se escriben con mayúsculas

Muchos de los compiladores actuales se rigen por el Standard ANSI X3J11 el cuál permite hasta 32 caracteres para los identificadores.

Los nombres de las variables deben ser cortos y han de reflejar para lo que la variable es usada, esto para facilitar recordarlas y simplificar el código. Deberán **iniciar por una letra minúscula** y pueden llevar combinaciones alfanuméricas entre mayúsculas y minúsculas y pueden tener carácter ‘_’ *underscore* para aclarar o simplificar el significado de la variable.

y suelen ir acompañadas de un comentario que especifique las unidades de la constante.



Se recomienda usar acrónimos, abreviaciones, mnemónicos; la letra mayúscula se usa para separar palabras dentro de la variable.

Algunas declaraciones se verán así:

```
signed int temp_interior; // definición para un sensor de temperatura
unsigned char velocidadActual; // definición para una variable velocidad
unsigned long tiempo_Act; // contador de tiempo actual
char keyBufIn[10] // buffer de entrada de teclado
```

Para el caso de declaración de apuntadores, estos deberán finalizarse con el acrónimo **Ptr** (ejemplo: bufferPtr), para indicar que se trata de un apuntador y no de un dato como tal. Esta recomendación puede aclarar bastante el uso de una variable en la mitad de un código, sin necesidad de recurrir a su definición.

6.8.4 Sobre las constantes

Las constantes deberán declararse con letras **MAYÚSCULAS** y pueden involucrar caracteres alfanuméricos acompañados del carácter ‘_’ underscore a fin de aclarar el significado de una constante.



Es útil acompañar la constante de un comentario, donde en primer lugar se especifiquen las unidades en las que se expresa la constante, con el objetivo de facilitar su modificación en un futuro que el código sea retomado por el programador original o por otro.

Tanto los paréntesis () como los corchetes { } han de cerrarse de inmediato para evitar errores en la compilación, que luego sea mas difícil ubicar la agrupación.



Ejemplos de declaración de constantes:

```
#define TEMPERATURA_MAX 85 // [oC] Temperatura
de Alarma
#define TIMEOUT_CONEXION 10000// [mSeg] Tiempo de espera de conexión
#define MAX_INTENTOS 4 // [intentos] máximos de entrar clave
```

6.8.5 Sobre los paréntesis y corchetes

Al digitar una estructura, y en general cada que se abre un paréntesis ‘(’ , corchete curvo‘{’ o un corchete recto ‘[’ , de inmediato se debe **CERRAR** y **TABULAR**, después de esto se digitará el código que va contenido entre ambos, evitando de esta forma que al programador se le olvide cerrarlo al final.

Así por ejemplo, si se construye un if.... else, lo primero que se debe digitar es su estructura completa, así:

```
if(){  
}else{  
}
```

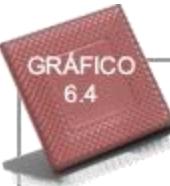


Y a continuación se digitará el contenido de cada uno de los bloques del if, del else y del condicional ‘()’...

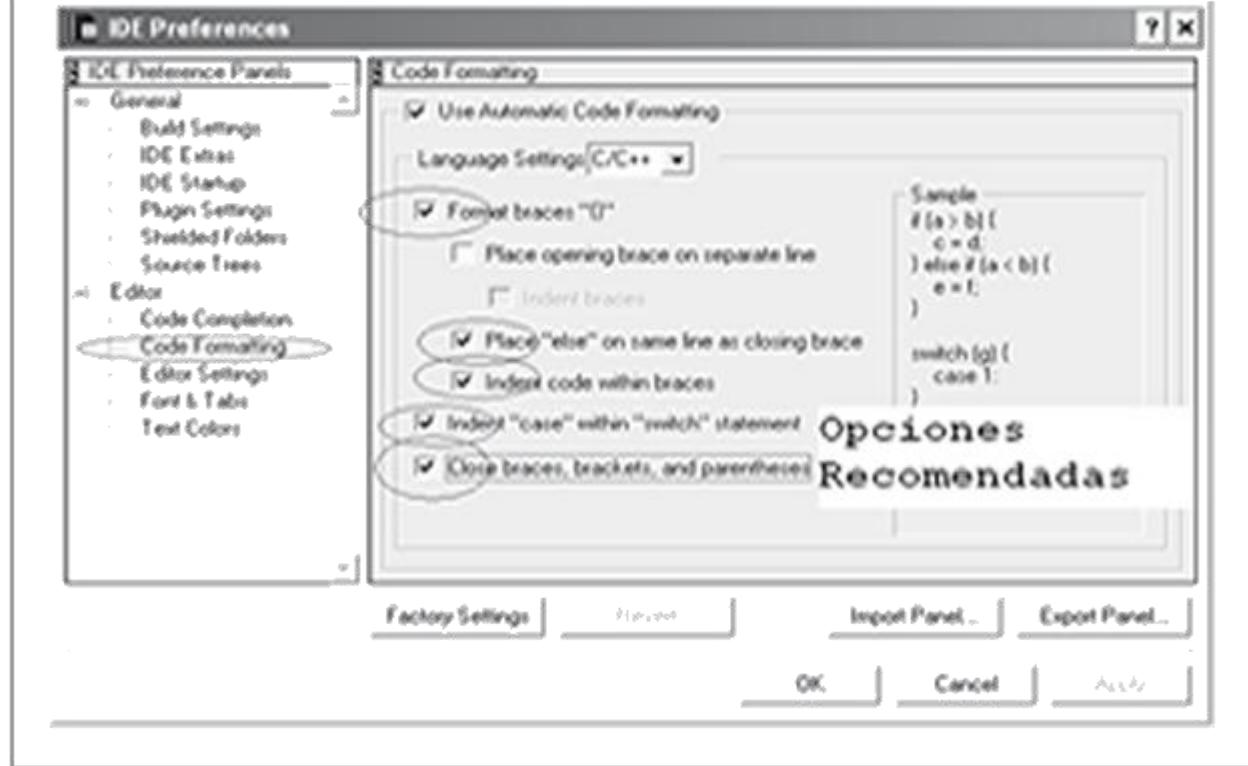
Existe una opción en el ambiente del Codewarrior® que permite cerrar estos caracteres de forma automática al digitar una apertura llamada **“Close braces, brackets, and parentheses”**, esta característica se habilita en **preferencias del IDE** ◊ **Code Formatting** (*ver Gráfico 6.4*).

En esta ventana se recomienda además marcar la opción **“Place “else” on the same line as closing brace”**, porque con ella se gana una línea de visibilidad en el editor y no es confuso si se realiza la tabulación adecuada, a su vez ésta se cubre seleccionando la opción **“Indent code within braces”** y el de **“Indent “case” within “switch” statement”**.

Estas opciones serán recordadas para todos los proyectos que se hagan en Codewarrior®, de tal forma que no es necesario hacerlo cada que se crea un nuevo proyecto.



Opciones recomendadas de edición de código en Codewarrior®.



Clíc para ver en alta resolución

RESUMEN DEL CAPÍTULO

La función es el componente principal de un proyecto modular, permite que el código pueda ser dividido y facilita la depuración.

A su vez, un grupo de funciones que hacen parte de un módulo particular, pueden compartir su funcionalidad a través de los prototipos que se publiquen en los archivos de cabecera (.H), que permiten a otros proyectos usar funciones ya implementadas y ensayadas.

El uso de apuntadores a funciones es útil en códigos que tienen procedimientos similares, junto con las secuencias do...while y for, permiten tener un código común que ejecuta diferentes procedimientos de tipo similar.

Las convenciones de programación facilitan la depuración, compartir código y el trabajo en equipo. Aunque no existe una convención definitiva si es recomendable usar alguna dentro de un proyecto o grupo de trabajo, especialmente en la declaración de variables, de funciones y la documentación.

PREGUNTAS Y EJERCICIOS PROPUESTOS

¿Cómo y cuándo decidi si los argumentos a una función se entregan por referencia o por valor?

¿Cómo calcular el nivel de recursividad (llamado sobre llamado) de una función en un proyecto en C? ¿Qué utilidad puede tener usar una forma convencional y estándar de programación en lenguaje C?

¿Qué desventaja puede existir al utilizar apuntadores a funciones en un proyecto desarrollado en C, en lugar de hacer el llamado directo a una función?

Realizar una función que retorna un valor booleano (FALSO o VERDADERO) si los 2 números querecibe como argumentos son números amigos. Se define como números amigos 2 números enteros **a** y **b**, tales que **a** es la suma de los divisores propios de **b** y **b** es la suma de los divisores propios de **a**. Implementar una función externa que encuentra 1 par de números amigos para el rango de 1 a 1000.

Implementar una función que retorna el estado del pin de IRQ del microcontrolador AP16A de la tarjeta AP-Link. Activar la salida OUT-1 de la tarjeta si el pin de IRQ está en bajo y apagarla una vez que el pin de IRQ esté en alto. NOTA: Para evitar daños en la tarjeta AP-Link, la prueba de este ejercicio deberá realizarse en un circuito externo al AP-Link, debido a que el pin de IRQ en la tarjeta está en el rango de 8 voltios.

Implementar la solución al siguiente problema de **Sudoku** usando una matriz de datos en C, la matriz inicial contiene ceros en las casillas que deben ser resueltas. Problema Sudoku: llenar las celdas de la cuadrícula 9x9 tal que cada fila, cada columna, y cada sub-cuadrícula contiene los números del 1 al 9.

3	9		7			5	2
6	1		2	9	4		
	7		8	4	1		6
2		9	3	8			4
4	8		1			6	9
7		9	4		5	8	
	3	8	5	7		4	
1	7		4		6		8
	2	8	6	3	7		

INTRODUCCIÓN

Las librerías del ANSI C proveen al diseñador una herramienta de programación adicional a las ya vistas con anterioridad.

El uso de estos grupos de funciones ya ensayadas, en las cuales solo se requiere incluir el respectivo archivo de cabecera (.H), y usarlas de forma adecuada, permite agilizar el diseño de forma notable; es por eso que se dedica este capítulo a la descripción individual de las librerías más comunes de cualquier compilador que cumpla ANSI, como lo son la librería de funciones matemáticas <math.h>, la estándar <stdlib.h>, la estándar de entrada/salida <stdio.h>, la de manejo de cadenas <string.h>, la de tipos de datos <ctype.h>, y la de manejo de tiempo <time.h>, cada una con un respectivo ejemplo práctico que fortalece el contenido teórico y genera confianza al diseñador para usar funciones que puedan resolver de forma más sencilla parte de sus algoritmos.

Sin embargo, en los sistemas embebidos y en especial en los de 8 bits, el costo de incluir librerías se paga caro, y es allí donde se invita al diseñador a medir de forma real el impacto en su sistema, y en algunos casos, a desarrollar librerías más específicas y óptimas.

En el mundo de los sistemas embebidos el uso y comercialización de librerías se está difundiendo cada vez más, haciendo que exista un número mayor de librerías que las discutidas en este capítulo; incluso, existen compañías de software dedicadas únicamente a la creación de librerías de nivel superior, como pueden ser las librerías para conexión a USB, librerías de manejo stack de TCP/IP que permiten la conexión a la web de un sistema embebido, librerías de generación de algoritmos complejos de FFT, de encriptado de datos y otras de manejo de dispositivos externos más específicos, como son los periféricos bluetooth, WIFI, zigbee, entre otros.

El completo entendimiento de este capítulo capacita al lector para poder usar cualquier librería que requiera incorporar en un futuro en su sistema embebido.

7.1 ¿QUÉ ES UNA LIBRERÍA ANSI C?

Las librerías del ANSI C consisten en *grupos de funciones* ya ensayadas y de las cuales solo se requiere incluir su respectivo archivo de cabecera (.H), y usarlas de forma adecuada.

La función de una librería opera como una caja de procesamiento “caja negra”, a la cual se le dan unos datos, y ella entrega un único resultado, denominado el retorno de la función (*ver Gráfico 7.1*).

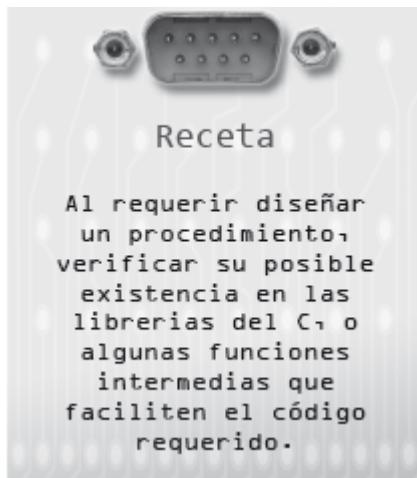
Para garantizar que la operación sea la adecuada, se debe conocer muy bien el prototipo de la función a usar, los tipos de datos que recibe con sus modificadores, de lo contrario, los resultados



Las librerías son herramientas de programación muy importantes, ya que permiten al diseñador utilizar funciones ya probadas e incluirlas en sus propios proyectos.



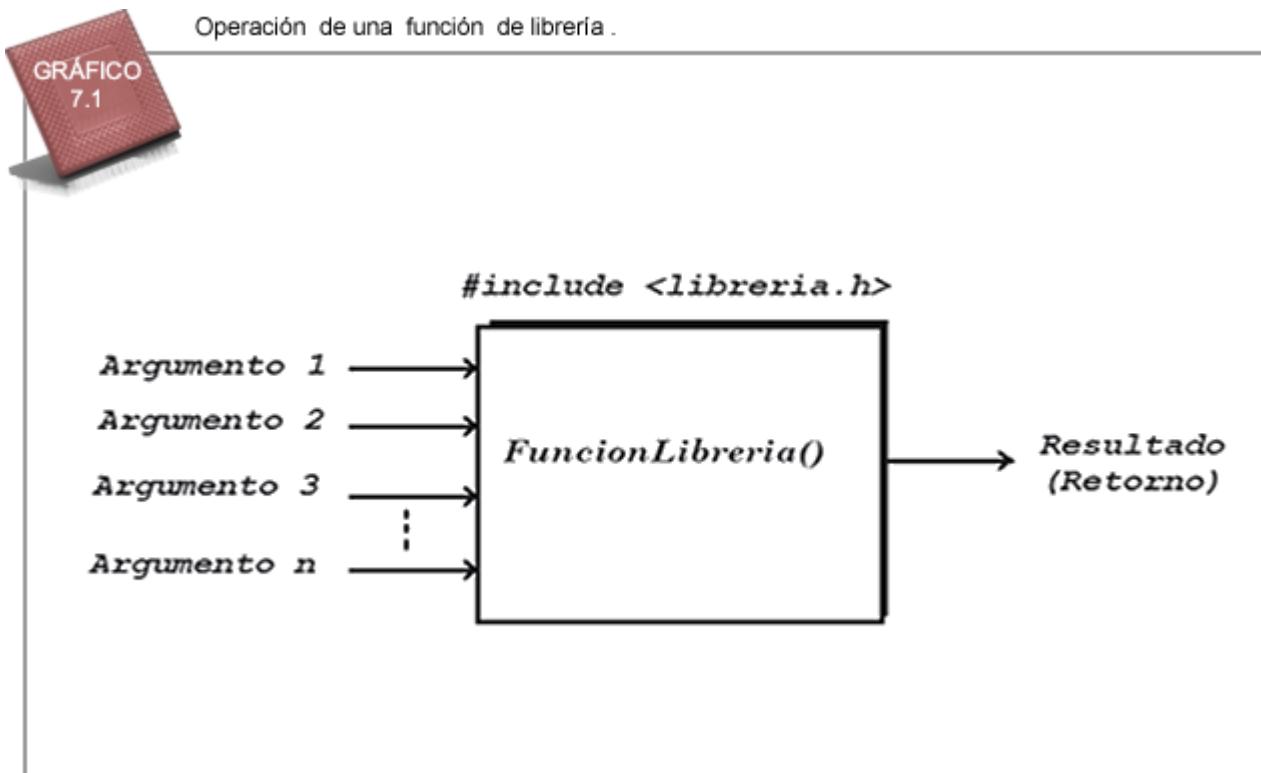
esperados pueden ser erróneos, provocar daños en la memoria o inclusive, que la función no retorne al flujo normal del programa.



Cuando se encuentran diferencias en los tipos de datos a operar, es responsabilidad del programador usar los moldes “cast” adecuados para que los datos recibidos y entregados coincidan.

Las librerías en C están divididas en grupos, dependiendo de su operación. Se resaltan acá los grupos más representativos, los más estándares y de uso más frecuente en el diseño de un sistema embebido.

Operación de una función de librería .



7.2 LIBRERÍA MATEMÁTICA <MATH.H>



No todas las funciones matemáticas provenientes de la librería <math.h> son eficientes, ya que algunos compiladores no pueden trabajar adecuadamente con aquellas desarrolladas con el sistema de punto flotante.



soporta impresión de números de nueve (9) dígitos. Esta limitación ocurre porque internamente se usa un unsigned long, el cual no puede contener más dígitos. El soporte de más dígitos incrementa la medida de las funciones de <math.h> y puede generar que la aplicación se ejecute de forma muy lenta. La lista de funciones disponibles de la librería <math.h> puede ser consultada en el Anexo # 2 y en la web.

Para usar las funciones que soportan números flotantes el proyecto en Codewarrior deberá tener seleccionada la opción de soporte de flotante (*ver Gráfico 7.2*).

La librería <math.h> incluye funciones que permiten realizar operaciones complejas de datos de tipo flotante o **double**, como lo son las funciones trigonométricas, logarítmicas y de potenciación.

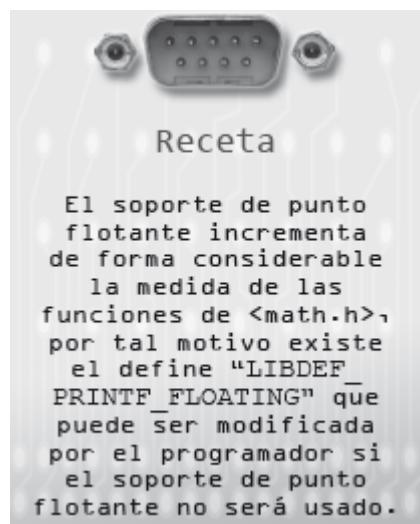
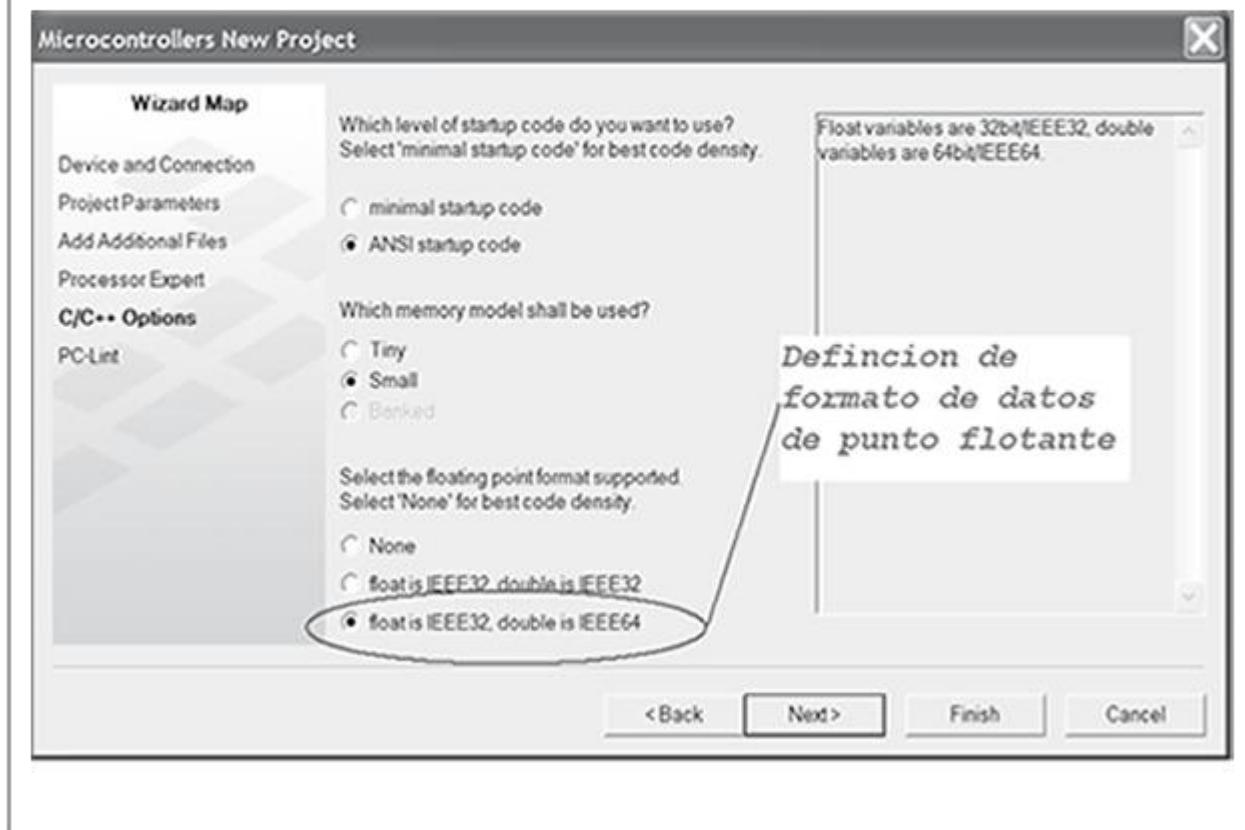
Algunas de las funciones no están implementadas en ciertos compiladores y máquinas que no soportan punto flotante, y por esta razón, en algunos casos, será necesario implementarlas por algún método numérico o acudir a la creación de una tabla de constantes sobre la que se busque el valor de resultado.

Un número flotante (o número real) a diferencia del entero, contiene una parte fraccional que puede ser de precisión variable dependiendo de la especificación en la que se soporte.





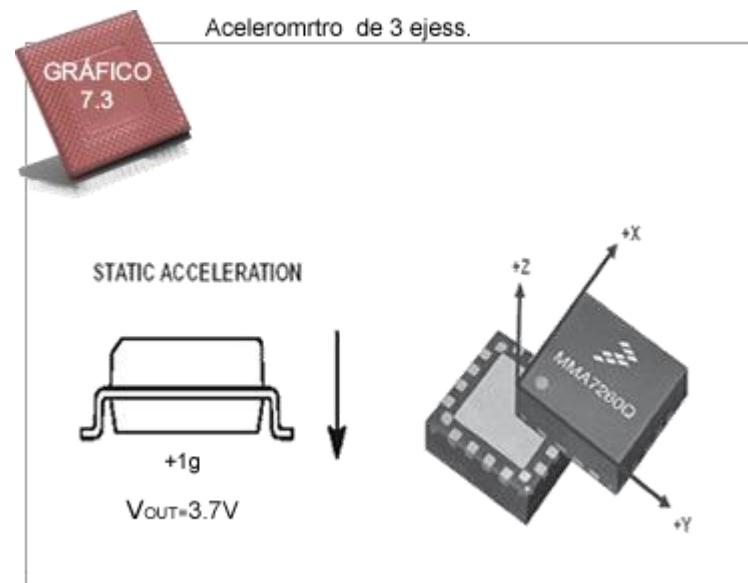
Opción de soporte de punto flotante en el Codewarrior.



El soporte de punto flotante incrementa de forma considerable la medida de las funciones de <math.h>, por tal motivo existe el define "LIBDEF_PRINTF_FLOATING" que puede ser modificada por el programador si el soporte de punto flotante no será usado.

Para ilustrar el uso de la librería math, se utilizará como accesorio un acelerómetro, que entrega una salida analógica correspondiente a la aceleración en cada eje X Y Z.

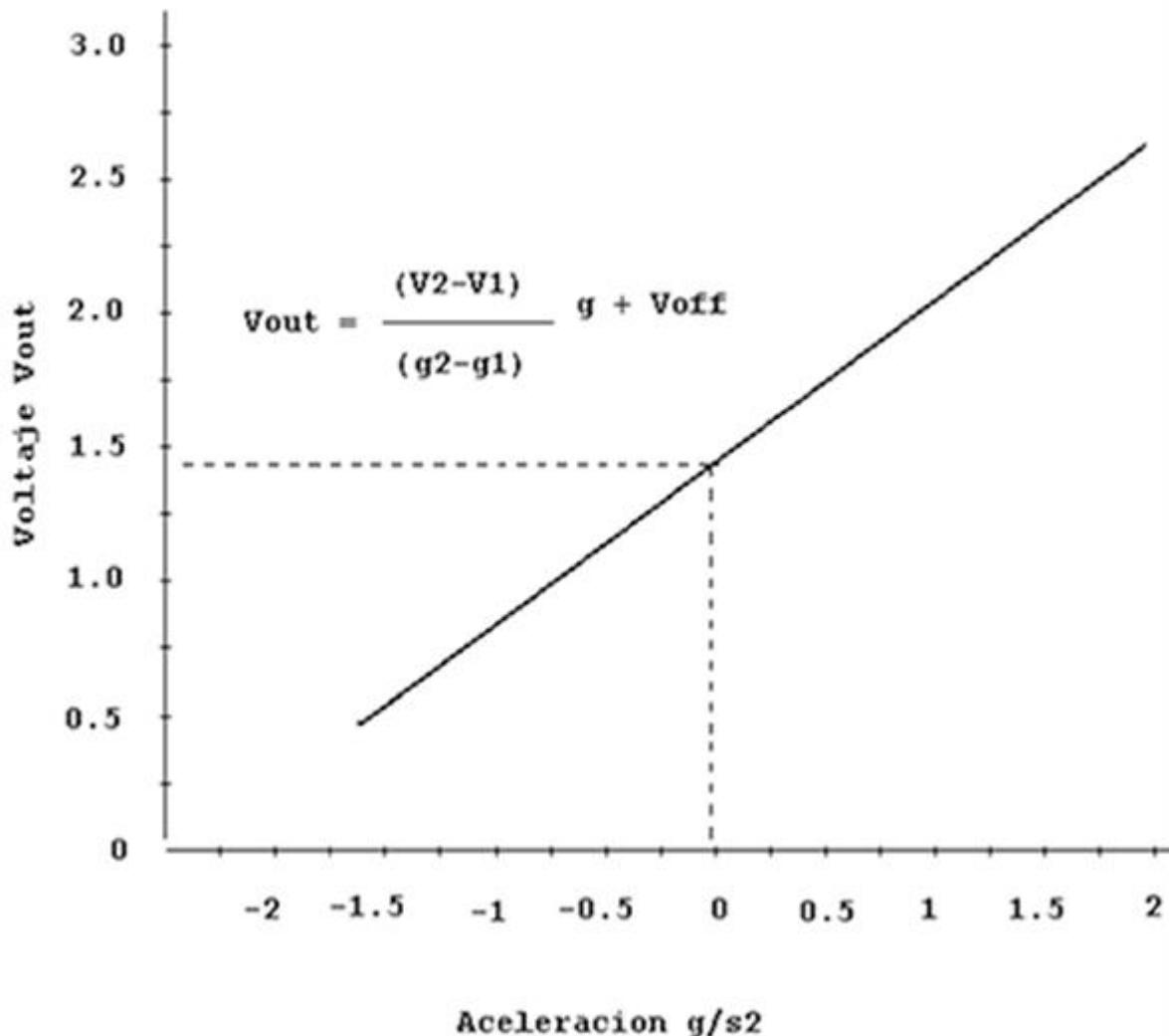
El acelerómetro usado será el MMA7260Q de Freescale™, éste proporciona una salida analógica con el valor correspondiente a la aceleración en cada uno de los ejes X Y Z.



La función de transferencia del acelerómetro que relaciona aceleración y voltaje de alimentación es la de una línea recta:



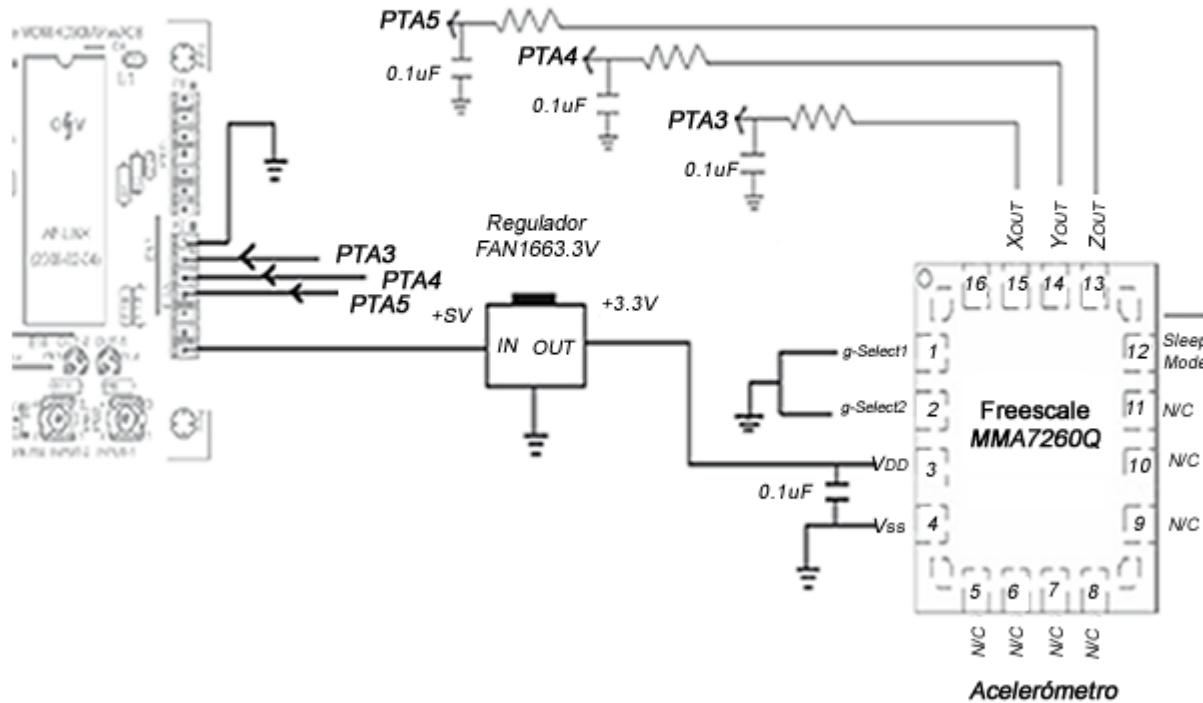
Función de transferencia del acelerómetro.





Interface del acelerómetro al microcontrolador AP16.

Sistema de Desarrollo
AP-LINK



EJEMPLO No. 26

ángulos de inclinación en grados

Realizar una rutina que retorne el ángulo de inclinación en grados de la tarjeta de evaluación, la cual contiene el acelerómetro MMA7260Q.

Indicar con una de las salidas que el ángulo de inclinación es mayor a 45 grados.

Solución:

Se parte de la siguiente ecuación, que involucra el voltaje de salida del acelerómetro y el ángulo de inclinación:

$$V_{out} = V_{off} + (\Delta V / \Delta g * g * \sin \theta)$$

Donde:

Vout : Voltaje de salida del acelerómetro

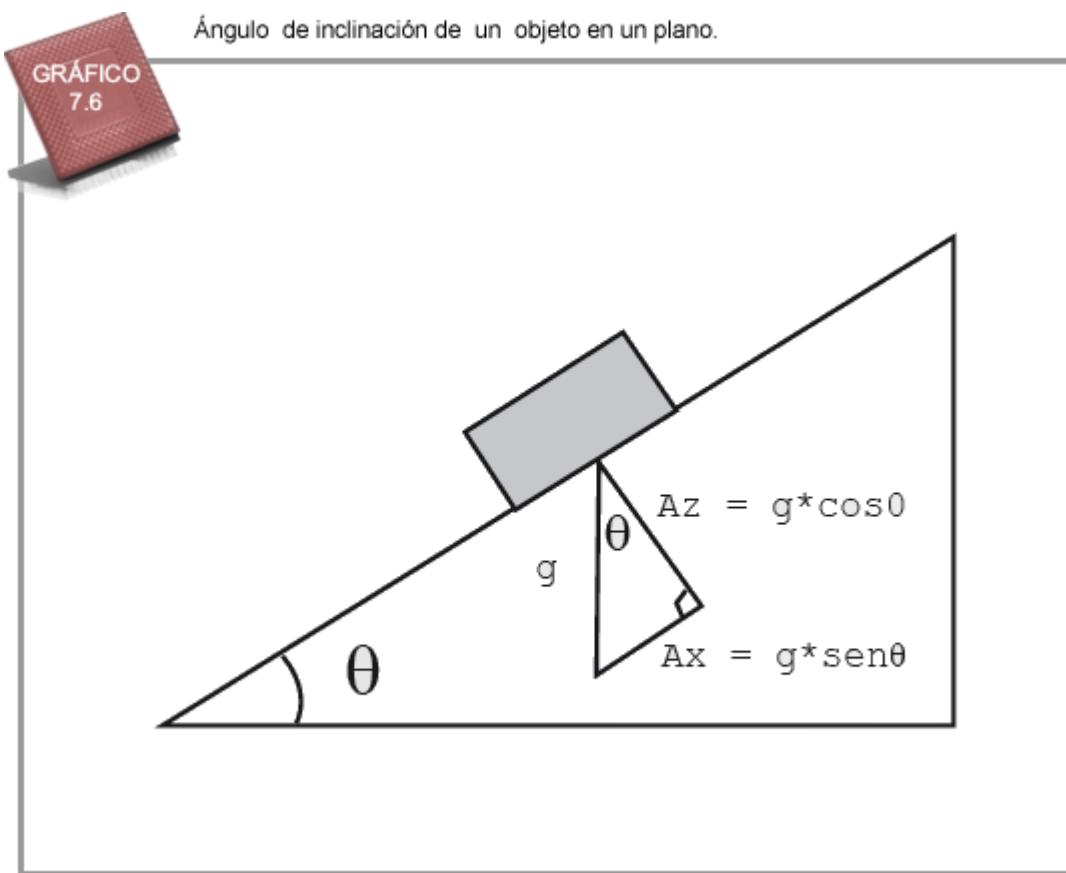
Voff : Voltaje de Offset

$\Delta V/\Delta g$: Sensibilidad

g : valor de la gravedad.

θ : ángulo de inclinación

La solución alternativa e utilizar geometría euclidiana para obtener el ángulo de inclinación en uno de los ejes:



El ángulo de inclinación en radianes en el plano Z, visto en 3 dimensiones X, Y y Z, estaría dado por la solución de la siguiente ecuación en el espacio:

$$\theta = \tan^{-1}(Ax / (\sqrt{Ay^2 + Az^2}))$$

Donde:

Ai : medida de la aceleración en el eje i.

PTA3 – ADC3 - Ax**PTA4 – ADC4 - Ay****PTA5 – ADC5 – Az**

```
***** Ejemplo 26 *****
//
// Angulo de Inclinación en grados
//
// Fecha: Feb 5,2009
// Asunto: Angulo de inclinacion
// Usando acelerometro MMA7260Q.
//
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// para Microcontrolador Freescale™ AP16.
// Version: 1.0 Por: Gustavo A. Galeano A.
//
*****



#include <hidef.h> /* for EnableInterrupts macro */
#include <math.h>
#include "derivative.h" /* include peripheral declarations */

#define EndConversionAdc() ADSCR_COCO
#define SelAdc3_Ax() ADSCR = 3
#define SelAdc4_Ay() ADSCR = 4
#define SelAdc5_Az() ADSCR = 5

#define Disable_COP() CONFIG1_COPD = 1 //deshabilita el COP

#define Led1On() PTC_PTC3 = 1; DDRC_DDRC3 = 1
#define Led1Off() PTC_PTC3 = 0; DDRC_DDRC3 = 1
#define NRO_PI (float)3.1415927 //equivalencia del número PI
//constantes de la función de transferencia Voltaje Vs gravedad
#define OFFSET (float)310
#define MAX_VAL (float)520
#define PENDIENTE (float)(MAX_VAL- OFFSET)

//variable usada para estados del Procedimiento ADC
char adcSt;
enum{
   INI_ADC,
   WAIT_AX,
   WAIT_AY,
   WAIT_AZ
};
//Inicialización del ADC del AP16A
void Adc_Init(void){
```

```

ADSCR_AIEN = 0; // modo de polling , no interrupción
ADSCR_ADCO = 0; // Solo una Conversión
ADICLK_ADIV2 = 0; //
ADICLK_ADIV1 = 0; //

ADICLK_ADIV0 = 0; // Clock Dividido por 1
ADICLK_ADICLK_BIT = 0; // External Clock
ADICLK_MODE1 = 0; //
ADICLK_MODE0 = 1; // Right Justified Mode
adcSt =INI_ADC;
}

unsigned int GetResultAdc(void){
    return ADR0;
}

struct dataAcel{
    unsigned int valX;
    unsigned int valY;
    unsigned int valZ;
};

struct datagAcel{
    float gx;
    float gy;
    float gz;
};

struct dataAcel datosAcel;
float anguloRad,magnitudD,anguloGrados,gTotal;
struct datagAcel acel;
unsigned long i;
//Procedimiento ADC, actualiza estructura con valores
void Adc_Run(void){
    switch(adcSt){
        case INI_ADC:
            SelAdc3_Ax();
            adcSt = WAIT_AX;
            break;
        case WAIT_AX:
            if(EndConversionAdc()){
                datosAcel.valX = GetResultAdc();
                SelAdc4_Ay();
                adcSt = WAIT_AY;
            }
            break;
        case WAIT_AY:
            if(EndConversionAdc()){
                datosAcel.valY = GetResultAdc();
                SelAdc5_Az();
                adcSt = WAIT_AZ;
            }
    }
}

```

```

        break;
    case WAIT_AZ:
        if(EndConversionAdc()){
            datosAcel.valZ = GetResultAdc();
            SelAdc3_Ax();
            adcSt = WAIT_AX;
        }
        break;
    }
}

//Función de conversión de radianes a grados
float Radianes2Grados(float rad){
static float grados;
    grados = (rad*(float)180)/NRO_PI;
    return grados;
}
void main(void) {
    Disable_COP();
    EnableInterrupts; /* enable interrupts */
    /* include your code here */
    Adc_Init();
    for(;;) {
        for(i=0;i<1500;i++) Adc_Run();

        // Angulo = tan-1(Ax/SQRT(Ay2 + Az2));

        acel.gx = ((float)datosAcel.valX - OFFSET)/PENDIENTE;
        acel gy = ((float)datosAcel.valY - OFFSET)/PENDIENTE;
        acel.gz = ((float)datosAcel.valZ - OFFSET)/PENDIENTE;
        gTotal = acel.gx+acel.gy+acel.gz;
        magnitudD = powf(acel.gy,(float)2);
        magnitudD += powf(acel.gz,(float)2);
        magnitudD= sqrtf(magnitudD);
        magnitudD = acel.gx/magnitudD;
        anguloRad = atanf(magnitudD);

        anguloGrados = Radianes2Grados(anguloRad);

        if(anguloGrados > 45){

            Led1On();

        }else{

            Led1Off();

        }
    }
}

```

```

    }

} /* loop forever */

/* please make sure that you never leave main */

}

```

Discusión:

En este ejemplo se inicializa el módulo ADC del AP16 para obtener conversiones a 10 bits de 3 de los canales de ADC que corresponden uno a uno a los valores de aceleración entregado por el acelerómetro, mediante la función **Adc_Init()**.

El procedimiento **Adc_Run()** mantiene la estructura datosAcel con los valores actualizados, sin embargo, para la solución de la ecuación, se requiere el valor de aceleración (g) de cada eje, la cual se hace con la función de transferencia de voltajeADC vs. valor de g en cada eje, valores que están en la estructura **acel**, con sus campos se realiza solución a la ecuación del ángulo, línea por línea, elevando a la potencia de 2 los campos **acel.gy** y **acel.gz**, invocando la función **powf()** de la librería math. Se saca la raíz cuadrada con la función **sqrtf()**, y se resuelve el ángulo en radianes, con el retorno de la función tan-1 (**atanf()**).

Materiales adicionales en la



Código fuente:
ángulos de inclinación
en grados, para
Freescale™.

Finalmente, y como parte de la solución al enunciado, se convierten los radianes a grados con

la implementación de la función **Radianes2Grados()**.

Las librerías matemáticas tienden a ocupar espacio considerable de memoria de programa y hacen gran uso del stack, por esta razón se recomienda ampliar el espacio de SP (**stack pointer**) y también considerar el tipo de operación, en caso que se requiera elevar un valor al cuadrado, se puede ahorrar más espacio de memoria realizando la multiplicación por sí misma, (**dato*dato**), en lugar de dato2 usando la función **powf**.

En algunos casos también puede resultar mejor acudir a una tabla de constantes en flash que realizar las operaciones matemáticas, en este caso la idea es mostrar el uso de las librerías.

7.3 LIBRERÍA ESTÁNDAR <STDLIB.H>

Esta librería proporciona funciones ya elaboradas que permiten convertir datos desde y hacia su valor numérico (entero o flotante), con su respectiva representación en caracteres ascII (*string*).



Las librerías <stdlib.h> aportan al diseñador funciones elaboradas para convertir datos desde y hacia su valor numérico, usando la memoria RAM de forma dinámica y de una manera más sencilla que si el programador tuviese que hacerlo de la forma convencional.

No es aconsejable utilizar las librerías <stdlib.h> en microcontroladores de poca RAM.

Las conversiones que facilitan estas funciones (**atoi**,**atof**, **atol**), son útiles en conversiones de datos que entran o salen del procesador en formatos que se puedan imprimir en un display LCD o en un terminal serial, pero que en determinado momento del procesamiento, estos datos requieren ser operados de forma matemática, en cuyo caso se requiere su valor numérico (cuantitativo) y no su representación ASCII (Anexo #1).

Provee además un grupo de funciones para el manejo dinámico de memoria (*malloc*, *alloc* y *free*), útil en procedimientos que requieren una zona determinada de **RAM** de forma temporal. Este tratamiento de la memoria **RAM** permite que sea manejada de forma óptima, ya que de la forma convencional de declaraciones estándar, solo se podría tener variables globales que consumen memoria continua de **RAM** o variables locales en el stack, que pierden su valor una vez la función que las declara es retornada.

En el manejo dinámico de memoria, es posible solicitar un número de bytes de la **RAM** para ser usados como memoria estática, que conservan su valor y no se ven alteradas por el llamado de funciones y/o movimiento del **stack**, pero una vez no sean requeridas por el software puede liberar su uso (función *free*), que permitirá que esas localidades de **RAM** puedan ser usadas por otros procedimientos posteriores, siempre y cuando los procedimientos sean excluyentes.

Algunas funciones de la librería `<stdlib.h>`, en especial las que están relacionadas con localización dinámica de memoria, requieren buena cantidad de **RAM** para poder funcionar, y es por esto que en procesadores de baja **RAM**, como es el caso del AP16A, tendrán limitaciones con estas funciones; sin embargo, se pueden realizar los ejemplos de estas funciones en modo **FULL_CHIP_SIMULATION** y seleccionando un procesador de mayor memoria, como lo es el Microcontrolador de Freescale™ **MC9S08GT60A**.

Materiales adicionales en la



Lista de funciones de la librería `<stdlib.h>` en el Anexo # 2, y explicación de cada función.

A continuación se realiza un ejemplo de localización dinámica usando las Funciones **malloc()** y **free()** donde se ilustra su uso.

EJEMPLO No. 27

localización dinámica de memoria

Objetivo:

Ilustrar el uso de las Funciones **malloc()** y **free()**, separando memoria para 3 buffers.
Usar el contenido de los buffers con valores de otras variables o constantes.

Solución:

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include <stdlib.h>

//definicion de los apuntadores a datos
char *buffer1Ptr;
char *buffer2Ptr;
char *buffer3Ptr;

//variables extras, para ilustrar uso de buffers
volatile unsigned char dato1=4,dato2=5,dato3=8;

void main(void) {
byte i; //variable local para los for()

    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    buffer1Ptr = malloc(5); //solicita 5 bytes
    buffer2Ptr = malloc(25); //solicita 25 bytes

    buffer1Ptr[0] = dato1; //usa los 5 bytes asignados
    buffer1Ptr[1] = dato2;
    buffer1Ptr[2] = dato3;
    buffer1Ptr[3] = dato2;
    buffer1Ptr[4] = dato1;
    free(buffer1Ptr); //libera los 5 bytes usados
    buffer2Ptr[0] = 0;
    buffer2Ptr[24] = 0xFF;

    free(buffer2Ptr);

    buffer3Ptr = malloc(14); //pide nuevamente 14 bytes

    for(i=0;i<14;i++) buffer3Ptr[i] = i+1; //usa el buffer

    free(buffer3Ptr); //libera el buffer

    buffer2Ptr = malloc(10); //solicita 10 bytes

    for(i=0;i<10;i++) buffer2Ptr[i] = i*5; //y los usa

    for(;;) {
        __RESET_WATCHDOG(); /* feeds the dog */
    } /* loop forever */
```

```
/* please make sure that you never leave main */  
}
```

Discusión:

Se incluye la librería **<stdlib.h>** para tener acceso a las Funciones de localización dinámica de memoria **malloc()** y **free()**.

En la línea:

buffer1Ptr = malloc(5);

Se invoca la función **malloc()** con argumento **5**, que le indica que se solicitarán **5 bytes de RAM** que se usarán de forma temporal y hasta que sea llamada la función **free()**. La función retorna un apuntador (o dirección) que es almacenada en **buffer1Ptr**, a partir de la cual se disponen de estos 5 bytes, lo que significa que se pueden usar: **buffer1Ptr[0], buffer1Ptr[1], buffer1Ptr[2], buffer1Ptr[3], buffer1Ptr[4]**.



Código fuente:
localización dinámica
de memoria, para
Freescale™.

En las líneas siguientes se solicitan **25 bytes**, cuya dirección de inicio es almacenada en **buffer2Ptr**, de este buffer se modifican los componentes inicial [0] y final [24], y se libera el buffer. Se solicita un nuevo espacio en memoria **RAM** para 14 bytes, cuyo

apuntador inicial es **buffer3Ptr** y allí se llena el buffer con la secuencia del 1 a 14, y el buffer es liberado con el uso de la función **free()**.

Nuevamente se solicitan 10 bytes para almacenar allí la secuencia: 0,5,10,15,20,25,30, 35,40,45, datos que permanecen intactos porque el buffer no es liberado.

Cada vez que se realiza una liberación de un espacio de memoria mediante la función **free()**, esta zona puede ser ocupada por la solicitud de otro requerimiento de memoria con la función **malloc()**, lo que indica que la memoria **RAM** puede ser compartida de forma dinámica por procedimientos que requieran una cantidad de memoria determinada de forma temporal.

En el caso de requerir memoria de forma definitiva resulta más eficiente que se realice la declaración de una variable global o estática, cuyo manejo es más rápido que a través de la librería **<stdlib.h>**.

7.4 LIBRERÍA ESTANDAR DE ENTRADA/SALIDA <STDIO.H>



Para recibir
y transmitir
datos de la
salida, la
librería

<stdio.h> aporta funciones con un formato establecido (sprintf) que facilita la visualización de la interfaz en algún dispositivo de salida.



La librería <stdio.h>: estándar input/output, contiene funciones para recibir y transmitir datos por el dispositivo estándar de entrada/salida (**getchar**, **gets**, **putchar**, **puts**, **printf**).

En programación de desktops estos dispositivos equivalen al teclado/monitor del PC, mientras que en sistemas embebidos y la gran mayoría de los compiladores, establece este dispositivo como el interfaz serial recepción/transmisión. (SCI o UART)¹.

Contiene funciones para escribir en una cadena de caracteres una variable con un formato establecido (**sprintf**), que facilita la visualización e interfaz con algún dispositivo de salida.

El siguiente ejemplo ilustra el manejo de cadenas que son ingresadas por un dispositivo de entrada (SCI).

Materiales adicionales en la



Lista de funciones de la librería <stdio.h> en el Anexo # 2, y explicación de cada función.

EJEMPLO No. 28

la calculadora serial

Objetivo:

Realizar un código en C que inicialice el SCI a 9600 Baudios 8N1, que reciba dos datos flotantes a ser operados.

Usar un PC con la aplicación estándar Hyperterminal para establecer comunicación con la tarjeta de desarrollo.

El teclado del PC será el dispositivo de entrada de datos seriales a la tarjeta y la pantalla del PC será usada como dispositivo de salida en la cual se muestran los datos que envía la tarjeta vía serial.

Operar los datos que se ingresan vía serial presentando los resultados en formato ASCII. El menú de configuración deberá presentarse así:

Operando 1 ?> Indicará que espera el operando 1
+ - * / % ?> Indica que espera la operación a realizar

Operando 2 ?> Indica que espera el operando 2

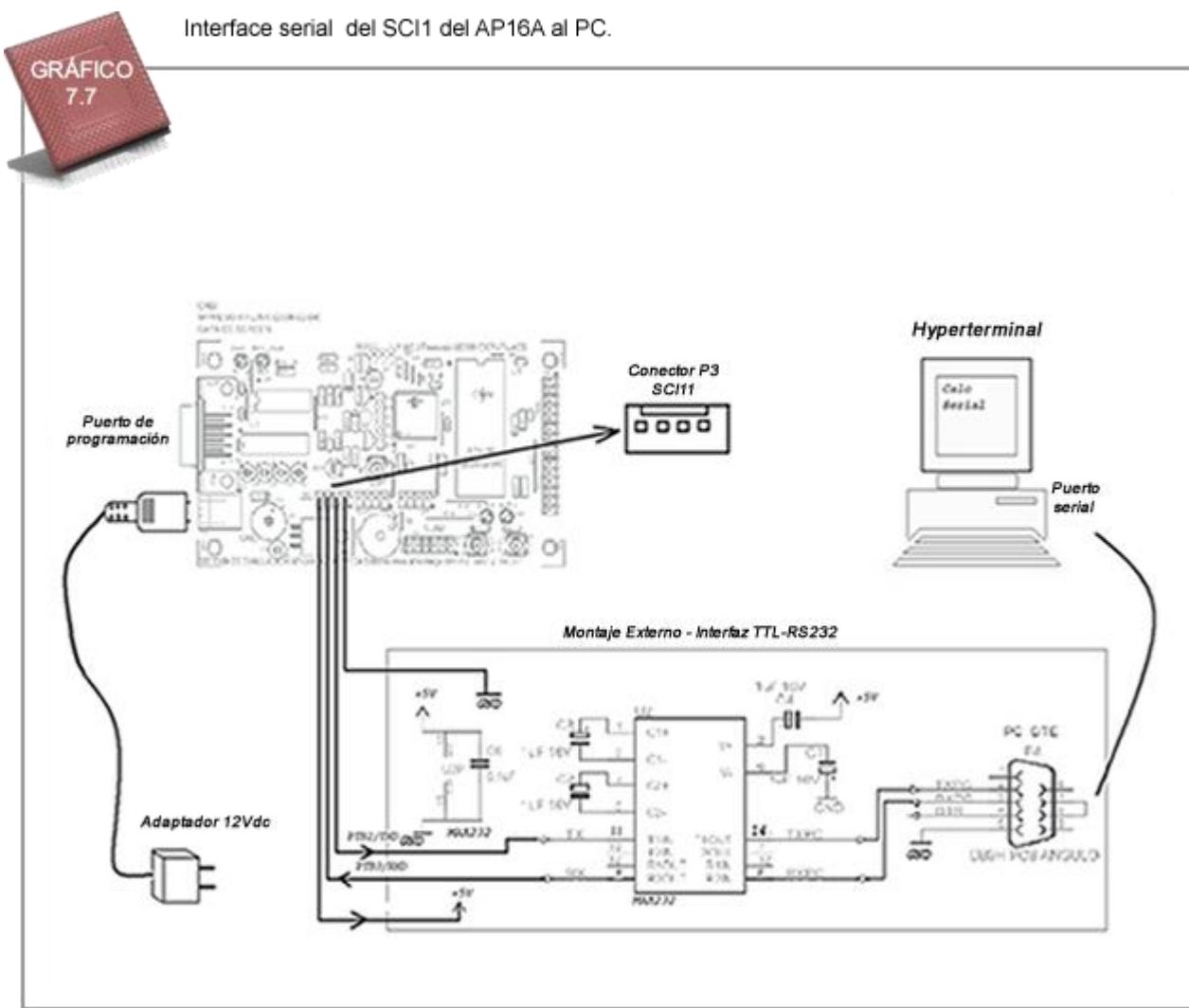
RESULTADO => Presenta el resultado

Si el cursor está esperando uno de los Operandos y se presiona <ENTER>, asumirá como Operando el valor del resultado anterior.

Solución :

Este ejemplo por defecto trabaja con la tarjeta de desarrollo PIC-Link, sin embargo

para el trabajo con la tarjeta AP-Link se requiere el montaje mostrado a continuación:



Clíc para ver en alta resolución

```
***** Ejemplo 28 *****
// La Calculadora Serial
// Fecha: Abril 1,2009
// Asunto: Diseño de Calculadora
// de datos flotantes vía serial
// Hardware: Sistema de desarrollo PIC-Link(2008-12-15)
// para Microcontrolador PIC 16F877A.
// Version: 1.0 Por: Gustavo A. Galeano A.
*****  

#define _FREESCALE_
#define _PIC_
```

```

#define __FREESCALE_
#include <hidef.h> /*for EnableInterrupts macro*/
#include "derivative.h" /*include peripheral declarations*/
#endif
#define __PIC_
#include "calcSerial.h"
#endif
#include <stdlib.h> //librería necesaria para el atof()
#include <stdio.h> //librería necesaria para el sprintf()
#define ENTER 0x0D //definición de constante retorno
#define LINE_FEED 0x0A //definición de constante avance
#define BUF_SIZE 20
byte buffer[BUF_SIZE]; //buffer para recepción y formato de resultados
double operando1,operando2,resultado;
char tipoOper;
/*Función de envío de carácter vía serial, recibe el carácter a enviar*/
void Serial_SendChar(byte Chr){
#define __FREESCALE_
    while(!SCS1_SCTE); //espera que este vacio
    SCDR = Chr; //envía dato al periférico SCI
    while(!SCS1_TC); //espera que se transmita el dato
#endif
#define __PIC_
    putc(Chr);
#endif
}
#define __FREESCALE_
void Serial_SendMsg(byte* Ptr){ /*Envía un string o cadena de caracteres*/
static byte *ptrData;
    ptrData = Ptr;
    while(*ptrData){
        Serial_SendChar(*ptrData); /* Envía el Dato */
        ptrData++;
    }
}
#endif
#define __PIC_
#define Serial_SendMsg printf
#endif
void Mcu_Init(void){ //Función Inicialización del MCU
#define __FREESCALE_
    /*
COPRS=0,LVISTOP=0,LVIRSTD=0,LVIPWRD=0,LVIREGD=0,SSREC=0,STOP=0,
COP
D=1 */
    CONFIG1: CONFIG1 = 0x01;
/* CONFIG2: STOP_ICLKDIS=0,STOP_RCLKEN=0,STOP_XCLKEN=0,OSCCLK1=0,
OSCCLK0=0,?? */

```

```

=0,??=0,SCIBDSRC=1 */
    CONFIG2 = 0x01;
#endif
#ifndef __PIC_
    setup_adc_ports(NO_ANALOGS);
    setup_adc(ADC_OFF);
    setup_psp(PSP_DISABLED);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC);
    setup_vref(FALSE);
    enable_interrupts(INT_TBE);
    enable_interrupts(INT_RDA);
#endif
}
void Serial_Init(void){ /*Inicialización del Serial a 9600,8N1*/
#ifndef __FREESCALE_
/* SCC1: LOOPS=0,ENSCI=0,TXINV=0,M=0,WAKE=0,ILTY=0,PEN=0,PTY=0 */
    SCC1 = 0x00; /* Configure the SCI */
/* SCBR: ??=0,??=0,SCP1=0,SCP0=0,??=0,SCR2=0,SCR1=1,SCR0=0 */
    SCBR = 0x02; /* Set the selected baud rate */
/* SCC2: SCTIE=0,TCIE=0,SCRIE=0,ILIE=0,TE=0,RE=0,RWU=0,SBK=0 */
    SCC2 = 0x00; /* Disable all interrupts */
    SCC1_ENSCI = 1; /* Enable device */
/* SCC3: ORIE=1,NEIE=1,FEIE=1,PEIE=1 */
    SCC3 |= 0x0F; /* Enable error interrupts */
    SCC2 |= ( SCC2_TE_MASK | SCC2_RE_MASK); /* Enable Tx, Rx*/
#endif
#ifndef __PIC_
    enable_interrupts(INT_TBE);
    enable_interrupts(INT_RDA)
#endif
}

//Función de recepción de operando vía serial
//Recibe el apuntador al que lleva el valor
void Serial_RxOperando(double *operandoPtr){
unsigned char i=0;
    do{
#ifndef __FREESCALE_
        while(!SCS1_SCRF); //Espera que llegue un carácter
        buffer[i] = SCDR; //Lo almacena en el buffer de Rx
#endif
#ifndef __PIC_
        buffer[i] = getch(); //Lo almacena en el buffer de Rx
#endif
        Serial_SendChar(buffer[i]); //ECO de la recepción
    }
}

```

```

        i++;
    }while((buffer[i-1]!= ENTER) && (i < (BUF_SIZE-1)));
    if((i==1) && (buffer[i-1]== ENTER)){
        *operandoPtr = resultado;
    }else{
        buffer[i-1] = 0; //fin de la cadena
        *operandoPtr = atof(buffer);
    }
}
void Serial_RxTipoOper(void){ /*Función de recepción del tipo de operador*/
    do{
#define __FREESCALE_
        while(!SCS1_SCRF); //Espera que llegue un caracter
        tipoOper = SCDR;
#endif
#define __PIC_
        tipoOper = getch();
#endif
        if(tipoOper == '+')||(tipoOper == '-')||(tipoOper == '*')||(tipoOper == '/')
            ||(tipoOper == '%')) Serial_SendChar(tipoOper); //ECO de la recepcion
    }while((tipoOper != '+') && (tipoOper != '-')
        && (tipoOper != '*') && (tipoOper != '/') && (tipoOper != '%'));
}
void CalculeResultado(void){ /*Función de cálculo del resultado*/
    switch(tipoOper){
        case '+': resultado = operando1+operando2;
        break;
        case '-': resultado = operando1-operando2;
        break;
        case '*': resultado = operando1*operando2;
        break;
        case '/': resultado = operando1/operando2;
        break;
        case '%':
            resultado = (double)((unsigned int)operando1%(unsigned int)operando2);
        break;
    }
}
void Presente_Resultado(void){ /*Formato y de envio de resultados*/
    (void)sprintf(buffer,"r\nRESULT ADO=> %f\r\n",resultado);
    Serial_SendMsg(buffer);
}
void main(void) { //Funcion principal
    Mcu_Init();
    Serial_Init();
    Serial_SendMsg("r\n**** ALFAOMEGA GRUPO EDITOR *****\r\n");
    Serial_SendMsg(" www.alfaomega.com.mx \r\n");
    Serial_SendMsg("---- Ejemplo Calculadora Serial ----\r\n");
}

```

```

for(;;){
    Serial_SendMsg("\r\nOperando1 ?> ");
    Serial_RxOperando(&operando1);
    Serial_SendMsg("\r\n+ - * / %% ?> ");
    Serial_RxTipoOper();
    Serial_SendMsg("\r\nOperando2 ?> ");
    Serial_RxOperando(&operando2);
    CalculeResultado();
    Presente_Resultado(); //envía resultado
}
}

```

Discusión:

El ejemplo sugiere conversiones de cadenas de caracteres que vienen por el puerto serial (valores ASCII), que corresponden a valores que deben ser convertidos a su respectivo valor flotante (ver Gráfico 7.8), para realizar las operaciones necesarias y presentar el resultado de la operación.

Por esta razón se incluye dentro de los archivos de cabecera la librería estándar: **<stdlib.h>**, que contiene el prototipo de la función que se requiere invocar **atof()**.

Se crea un arreglo global llamado **buffer**, usado para el almacenamiento de los datos de las cadenas que son ingresados por el periférico serial SCI, y para la conversión del resultado de la operación en una cadena de caracteres que será enviada de regreso al interfaz serial con el resultado de la operación de los datos; en esta última conversión se requiere cambiar el valor flotante a una cadena de valores ASCII que lo representa, en este caso se usará la librería **<stdio.h>**, y su función de conversión **sprintf()**.

El programa principal, **main**, realiza las inicializaciones y envía los mensajes de de presentación, y queda en un loop continuo que realiza las siguientes Funciones:

- Envía mensaje de solicitud del Operando 1 → **Operando1 ?>**
- Recibe y valida el Operando1, que es almacenado en la variable **operando1**
- Envía mensaje de solicitud del tipo de operación → + - * / % ?>
- Recibe y valida el tipo de operación, que es almacenado en la variable **tipo Oper**
- Envía mensaje de solicitud del Operando 2 → **Operando2 ?>**
- Recibe y valida el Operando1, almacenado en la variable **operando2**
- Realiza el cálculo del resultado que resulta de **operando1 tipoOper operando2**
- Envía el valor de la variable resultado por el puerto serial, previa conversión de su valor a su formato ASCII.

Desde un interfaz serial (ejemplo Hyperterminal de windows) a 9600 Baudios 8N1 (8 bits, No paridad y 1 bit de stop), permitirá al usuario ingresar y visualizar los resultados como lo muestra el *Gráfico 7.8*.

Ejemplo calculadora serial: Hyperterminal a 9600 Baudios 8N1.



```
CalcSerial - HyperTerminal
File Edit View Cell Transfer Help
D S 3 0 2 1 2

---- ALFAOMEGA GRUPO EDITOR -----
www.alfaomega.com.mx
---- Ejemplo Calculadora Serial ----

Operando1 ?> 34.21
* - * / % ?> *
Operando2 ?> 45.23
RESULTADO=> 1547.318304

Operando1 ?>
* - * / % ?> *
Operando2 ?> 2
RESULTADO=> 3094.636604

Operando1 ?> 9345
* - * / % ?> -
Operando2 ?> 50
RESULTADO=> 9295.000049

Operando1 ?>
```

Connected 0:00:54 Auto detect 9600 8-N-1 CROLL CAPS NUM Capture Print echo

Materiales adicionales en la



Código fuente: la calculadora serial, para Freescale™ y Microchip™.

Las funciones de apoyo realizan los procedimientos internos:

void Serial_SendMsg(byte* Ptr) → Envía el mensaje apuntado por **Ptr** vía serial, usando la sentencia for, que envía caracteres a partir de **Ptr** y hasta que encuentre un valor nulo.

void Serial_RxOperando(double *operandoPtr) → Recibe una cadena de caracteres vía serial y almacena su valor flotante en la dirección apuntada por **operandoPtr**.

void Serial_RxTipoOper(void) → Recibe y actualiza el valor de la variable **tipoOper**, previa validación del carácter ingresado.

void CalculeResultado(void) → Actualiza el valor (double) de la variable resultado, usando una sentencia **switch** para determinar la operación solicitada por el usuario de la calculadora.

void Presente_Resultado(void) → Convierte el valor que está en la variable resultado a su representación con formato en ASCII y almacenado en **buffer**, para ser enviado vía serial.

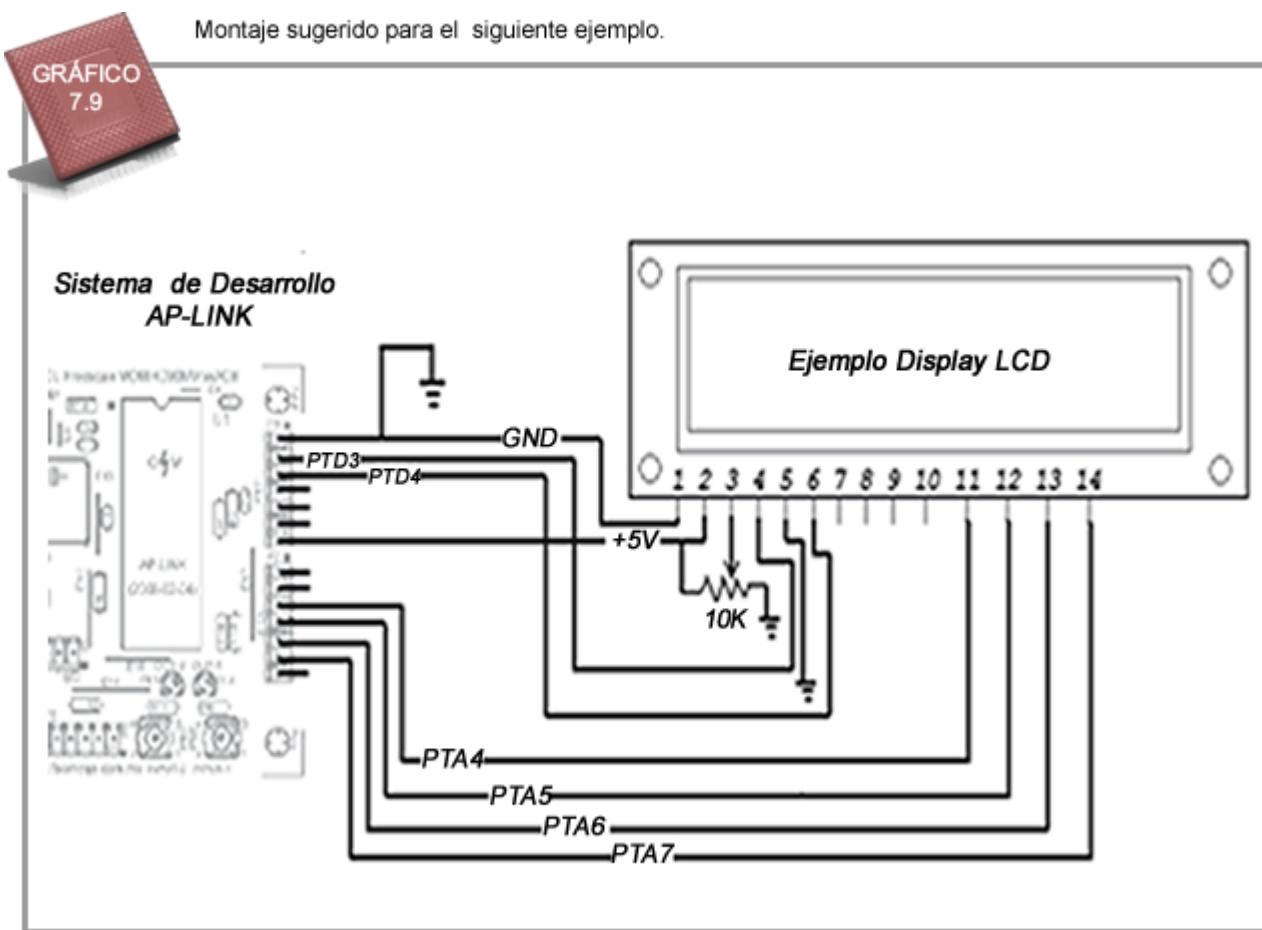
7.5 Librería de manejo de cadenas <string.h>



Materiales adicionales en la web
Lista de funciones de la librería <string.h> en el Anexo # 2, y explicación de cada función.

Permite la medida (**strlen**) comparación (**memcmp,strcmp**), movimiento (**memmove**) y duplicación (**memcpy,strlen**) de datos o cadenas de caracteres. Contiene funciones específicas para ubicación de caracteres dentro de una secuencia de datos o cadena (**memchr,strchr**) y conversión de formatos de tiempo (**tm vs. struct_tm**) para manejo de RTC (**Real Time Clock**), entre otras funciones de manejo de bloques de datos y cadenas de caracteres (**strings**).

Montaje sugerido para el siguiente ejemplo.



EJEMPLO No. 29

Manejo de display LCD

Objetivo:

Realizar un código en C que maneje un display LCD 2 filas x 20 columnas. Utilizar el módulo Serial, para enviar al LCD varios comandos:

mensaje: permite recibir un nuevo mensaje a imprimir en el LCD.

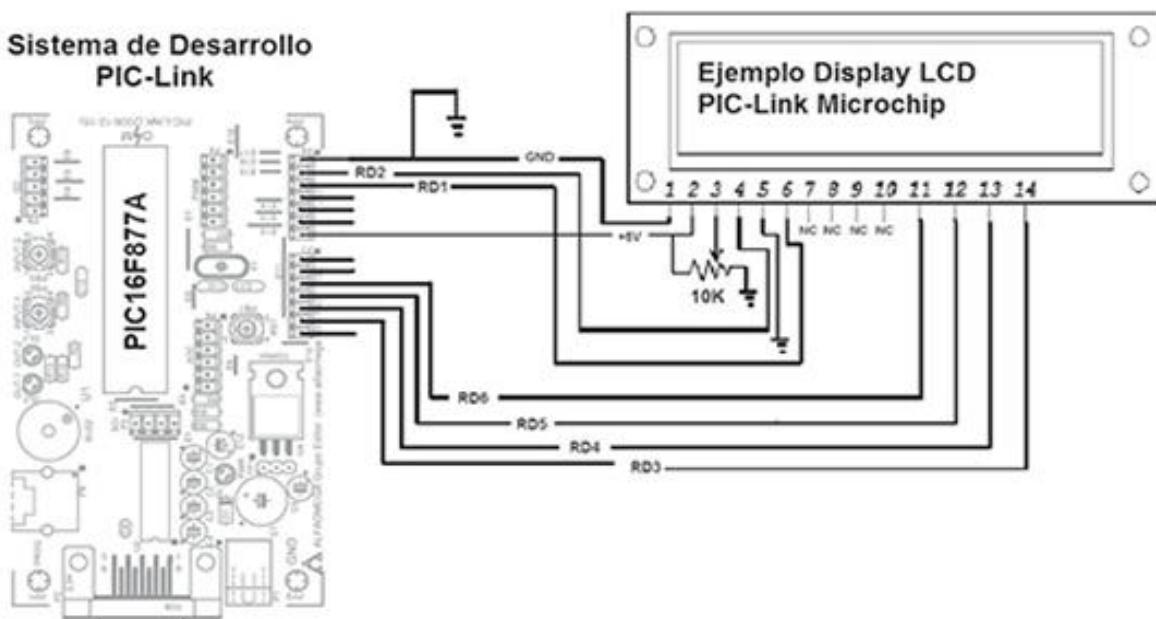
rotar: debe rotar el mensaje de izquierda a derecha.

letras: imprime el mensaje ingresando las letras una a una de izquierda a derecha.

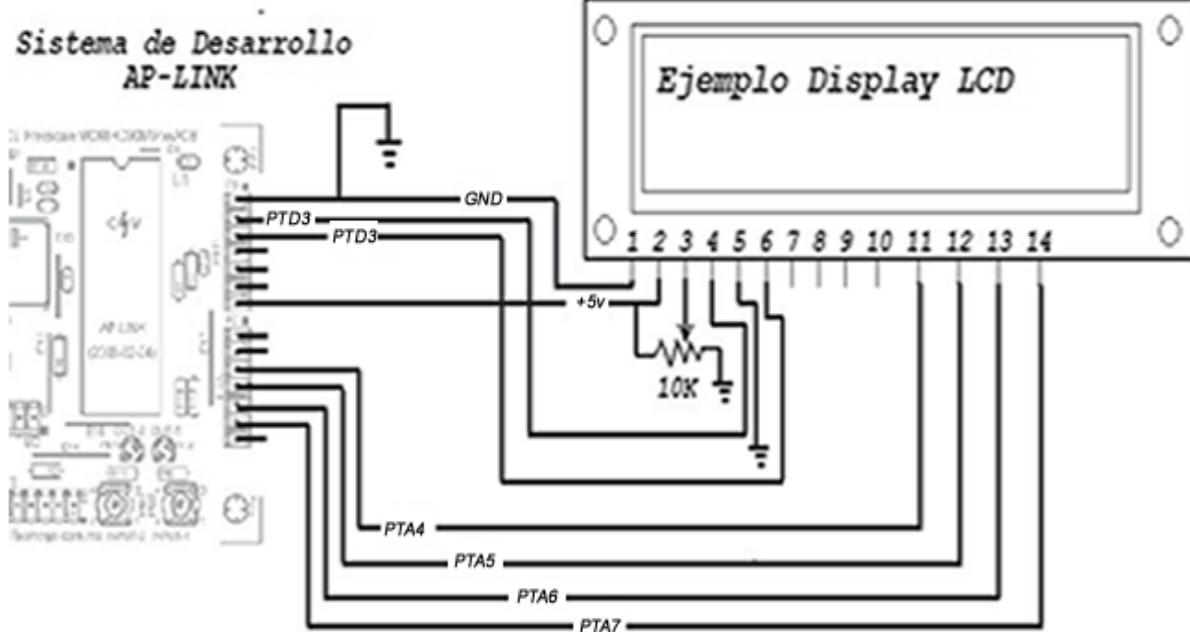
palabras: imprime una a una las palabras del mensaje.

Solución:

Montaje sugerido para PIC-Link Microchip™.



Montaje sugerido para AP-Link Freescale



```
//***** Ejemplo 29 *****
```

```
// Manejo de display LCD
```

```
// Fecha: Abril 2,2009
```

```
// Asunto: Código que recibe comandos vía
```

```
// serial y los ejecuta en un LCD 2x2.
```

```
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
```

```
// o PIC-Link(2008-12-15)
```

```
// Versión: 1.0 Por: Gustavo A. Galeano A.
```

```
//*****
```

```
#undef __FREESCALE_
```

```
#define __PIC_
```

```
#ifdef __PIC_
```

```
#include "displayLcd.h"
```

```
#include "lcd.c"
```

```

#include "serial.c"

#endif

#ifndef __FREESCALE__

#include <hidef.h> /* for EnableInterrupts macro */

#include "derivative.h" /* include peripheral declarations */

#include "lcd.h" //incluye funciones de manejo de LCD

#endif

#include "serial.h" //incluye funciones de comunicación SERIAL

#include <string.h> /*incluye funciones de libreria string.h*/

void SetMensajeLcd(char *msgPtr);

void EfectoRotar(char * msgPtr);

void EfectoLetras(char *msgPtr);

void EfectoPalabras(char *msgPtr);

#define NRO_COMANDOS 4

#ifndef __FREESCALE__

typedef void vFuncPtrCharPtr(char*); // Definición de la estructura de comandos
y funciones

struct komandosFx{

    char *mensajePtr;
    vFuncPtrCharPtr *funcionPtr;
};

struct komandosFx comandosFx[NRO_COMANDOS]={

    "mensaje", SetMensajeLcd,
    "rotar", EfectoRotar,
};

```

```

“letras”, EfectoLetras,
“palabras”, EfectoPalabras,
};

#endif

byte mensajeLcd[BUF_SIZE] = "Ejemplo Manejo LCD";
char buffer[BUF_SIZE];
char mensajeAlfa[] = "*** Alfaomega **** ";
char mensajeGrup[] = " Grupo Editor ";
void Mcu_Init(void){ /*Inicialización del registros del MCU*/
#define __FREESCALE__
/*CONFIG1:
COPRS=0,LVISTOP=0,LVIRSTD=0,LVIPWRD=0,LVIREGD=0,SSREC=0,STOP=0,
COP D=1 */
CONFIG1 = 0x01;

/*CONFIG2:
STOP_ICLKDIS=0,STOP_RCLKEN=0,STOP_XCLKEN=0,OSCCLK1=0,OSCCLK0=0,
?? =0,??=0,SCIBDSRC=1 */
CONFIG2 = 0x01;

#endif
#define __PIC__
setup_adc_ports(NO_ANALOGS);
setup_adc(ADC_OFF);
setup_psp(PSP_DISABLED);
setup_spi(SPI_SS_DISABLED);
setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);

```

```

setup_timer_1(T1_DISABLED);

setup_timer_2(T2_DISABLED,0,1);

setup_comparator(NC_NC_NC_NC);

setup_vref(FALSE);

#endif

}

void SetMensajeLcd(char *msgPtr){ /*Recepción nuevo mensaje vía SERIAL*/
    Serial_SendMsg("\r\n Mensaje LCD>");
    Serial_RecibirCmd(msgPtr);
    Lcd_Cls();
    PutsD(msgPtr);
    Serial_SendMsg("\r\n");
}

void EfectoRotar(char * msgPtr){ /*Rotar el mensaje en el LCD*/
    unsigned char i;
    char msgActual[40]="" ;
    Serial_SendMsg("\r\nRotando Mensaje... ");
    Lcd_Cls();
    msgPtr[19] = 0;
(void)strcat(msgActual,msgPtr);
    for(i=0;i<40;i++){
        Lcd_Cls();
        PutsD(msgActual+i);
        Delay_Lcd(20000);
    }
}

```

```

}

Serial_SendMsg("\r\nFin Rotacion.");

Serial_SendMsg("\r\n");

}

void EfectoLetras(char *msgPtr){ // Función de Efecto letra por letra

unsigned char i,j,caracter;

Serial_SendMsg("Efecto letras");

Lcd_Cls();

i = (unsigned char)strlen(msgPtr); //completa con espacios

while(i<20){ msgPtr[i] = ' '; i++; }

for(j=0;j<19;j++){

    caracter = msgPtr[j];

    for(i=20;i!=j;i--){

        Lcd_Goto(i-1);

        PutchD(msgPtr[j]);

        PutchD(' ');

        Delay_Lcd(10000);

    }

    Delay_Lcd(10000);

}

Serial_SendMsg("\r\nFin Efecto Letras");

Serial_SendMsg("\r\n");

}

void EfectoPalabras(char *msgPtr){ // Función de Efecto de palabra por palabra

```

```
en el LCD

unsigned char i;

char *tempPtr,*topPtr,*msgPtrBck,finMsg;

Serial_SendMsg("Efecto palabras");

msgPtrBck = msgPtr;

msgPtrBck[19] = 0;

i = (unsigned char)strlen(msgPtrBck);

finMsg = i;

msgPtrBck[i] = ' ';

tempPtr = msgPtrBck;

topPtr = msgPtrBck + i;

Lcd_Cls();

while(tempPtr <= topPtr){

    tempPtr = strchr(tempPtr, ' ');

    if(tempPtr){

        tempPtr[0] = 0;

        PutsD(msgPtrBck);

        PutchD(' ');

        tempPtr[0] = ' ';

        Delay_Lcd(50000);

        tempPtr++;

        msgPtrBck = tempPtr;

    }

}
```

```

msgPtr[finMsg] = 0;

Serial_SendMsg("\r\nFin Efecto palabras");

Serial_SendMsg("\r\n");

}

#ifndef __PIC__

char mensajeCmd[]="mensaje";

char rotarCmd[]="rotar";

char letrasCmd[]="letras";

char palabrasCmd[]="palabras";

#endif

void EjecuteCmd(char *bufferCmd){ /*Ejecución de una cadena*/

#ifndef __FREESCALE__

unsigned char i=0,comandFound=0;

while((!comandFound) && (i<NRO_COMANDOS)){

if(!strcmp(bufferCmd,comandosFx[i].mensajePtr)){

comandFound = 1;

comandosFx[i].funcionPtr(mensajeLcd);

}

i++;

}

#endif

#endif __PIC__

if(!strcmp(bufferCmd,mensajeCmd)){

SetMensajeLcd(mensajeLcd);

```

```

}else{
    if(!strcmp(bufferCmd,rotarCmd)){
        EfectoRotar(mensajeLcd);
    }else{
        if(!strcmp(bufferCmd,letrasCmd)){
            EfectoLetras(mensajeLcd);
        }else{
            if(!strcmp(bufferCmd,palabrasCmd)){
                EfectoPalabras(mensajeLcd);
            }
        }
    }
#endif
}
void main(void) { // Función Principal del programa
    Mcu_Init(); //inicializa microcontrolador
#ifndef __PIC_
    delay_ms(2000);
#endif
    Serial_Init(); //inicializa modulo SERIAL
    Lcd_Init(); //inicializa modulo LCD
    PutsD(mensajeAlfa);
    Lcd_Goto(20);
    PutsD(mensajeGrup);
    Serial_SendMsg("\r\n***** ALFAOMEGA GRUPO EDITOR *****\r\n");
    Serial_SendMsg(" www.alfaomega.com.mx \r\n");
    Serial_SendMsg("----- Ejemplo Display LCD ----- \r\n");
    for(;; {
        Serial_RecibirCmd(buffer);
        EjecuteCmd(buffer);
    } /* loop forever */
    /* please make sure that you never leave main */
}
/*===== Modulo LCD =====*/
/*Distribución de los pines del LCD
1.GND 2.Vcc 3.Contraste (Pot 10K GND y VCC)
4. RS dato (el pin en 1) o instruc (comando LCD) (el pin en 0)
5.WE conectado a GND 6.E enable
7. D0 NC 8. D1 NC 9.D2 NC 10.D3 NC
11.D4 12.D5 13.D6 14.D7*/
#define __FREESCALE_
#ifndef __FREESCALE_
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "lcd.h" // incluye prototipos de funciones de LCD
#define Inst() PTD_PTD3 = 0; DDRD_DDRD3= 1; Delay_Lcd(20)

```

```

#define dato() PTD_PTD3 = 1; DDRD_DDRD3 = 1; Delay_Lcd(20)
#define E1() PTD_PTD4 = 1; DDRD_DDRD4 = 1; Delay_Lcd(20)
#define E0() PTD_PTD4 = 0; DDRD_DDRD4 = 1; Delay_Lcd(20)
#define Data4Low() PTA_PTA4 = 0; DDRA_DDRA4 = 1
#define Data5Low() PTA_PTA5 = 0; DDRA_DDRA5 = 1
#define Data6Low() PTA_PTA6 = 0; DDRA_DDRA6 = 1
#define Data7Low() PTA_PTA7 = 0; DDRA_DDRA7 = 1
#define Data4High() PTA_PTA4 = 1; DDRA_DDRA4 = 1
#define Data5High() PTA_PTA5 = 1; DDRA_DDRA5 = 1
#define Data6High() PTA_PTA6 = 1; DDRA_DDRA6 = 1
#define Data7High() PTA_PTA7 = 1; DDRA_DDRA7 = 1
#endif
#ifndef __PIC_
#define Inst() output_low(PIN_D2); delay_us(10)
#define dato() output_high(PIN_D2); delay_us(10)
#define E1() output_high(PIN_D1); delay_us(10)
#define E0() output_low(PIN_D1); delay_us(10)
#define Data4Low() output_low(PIN_D6)
#define Data5Low() output_low(PIN_D5)
#define Data6Low() output_low(PIN_D4)
#define Data7Low() output_low(PIN_D3)
#define Data4High() output_high(PIN_D6)
#define Data5High() output_high(PIN_D5)
#define Data6High() output_high(PIN_D4)
#define Data7High() output_high(PIN_D3)
#endif
#define RETURN_HOME 0x02 // Comandos para el LCD
#define FUNCTION_SET 0x28
#define CLEAR_LCD 0x01
#define DISPL_ON_OFF 0x06
static void SendData(unsigned char data);
#ifndef __FREESCALE_
void Delay_Lcd(unsigned int timeDly){ /*Función retardo para LCD*/
    unsigned int i;
    for(i=0; i<timeDly;i++);
}
#endif
#ifndef __PIC_
void Delay_Lcd(unsigned int16 msegs){ /*Función retardo para LCD*/
    #ifndef __PIC_
        delay_ms(msegs/100);
    #endif
}
#endif
static void LcdInst(unsigned char instr){ /*Envío de instrucción al LCD*/
    Inst();
    SendData(instr);
}

```

```

void Lcd_Cls(void){ /*Funcion de limpiar el LCD*/
LcdInst(0x01);
}
void PutchD(unsigned char data){ /*impresión de un caracter en el LCD*/
    dato();
    SendData(data);
}
void PutsD(char *s){ /*impresión de un mensaje, recibe un apuntador*/
    char *msgPtr;
    Delay_Lcd(500);
    msgPtr = s;
    while(*msgPtr){
        PutchD(*msgPtr);
        msgPtr++;
    }
}
void Lcd_Goto(unsigned char pos){ /*posicionamiento en el LCD 2x20*/
    Delay_Lcd(1000);
    if(pos < 20) LcdInst(0x80+pos);/*envía comando para ubicar el cursor*/
    else LcdInst(0xC0+pos-20);
}
void Lcd_Init(void){ /*Función de Inicialización del LCD*/
    Delay_Lcd(10000); /*Retardo inicial de encendido*/
    Data4Low();
    Data5Low();
    Data6Low();
    Data7Low();
    LcdInst(RETURN_HOME);
    Delay_Lcd(1500);
    LcdInst(RETURN_HOME);
    Delay_Lcd(1500);
    LcdInst(FUNCTION_SET);
    Delay_Lcd(1000);
    LcdInst(FUNCTION_SET);
    Delay_Lcd(1000);
    LcdInst(0x0C);
    Delay_Lcd(2000);
    LcdInst(DISPL_ON_OFF); /*Display ON OFF No cursor no Blink*/
    Delay_Lcd(2000);
    LcdInst(CLEAR_LCD); /*Clear and Home*/
    Delay_Lcd(2000);
}

static void SendData(unsigned char data){ /*Envío de instrucción o dato*/
    E1();
}

```

```

if(data & 0x80){ Data7High();} else { Data7Low(); }
if(data & 0x40){ Data6High();} else { Data6Low(); }
if(data & 0x20){ Data5High();} else { Data5Low(); }
if(data & 0x10){ Data4High();} else { Data4Low(); }
Delay_Lcd(200);
E0();
Delay_Lcd(200);
E1();
if(data & 0x08){ Data7High();} else { Data7Low(); }
if(data & 0x04){ Data6High();} else { Data6Low(); }
if(data & 0x02){ Data5High();} else { Data5Low(); }
if(data & 0x01){ Data4High();} else { Data4Low(); }
Delay_Lcd(200);
E0();
}
//***** SERIAL.C *****
// Codigo fuente SERIAL.C
// Abril 2,2009
// Asunto: Modulo de Manejo del SCI
// Hardware: Sistema de desarrollo AP-Link(2008-01-14)
// o PIC-Link (2008-12-15)
// Version: 1.0 Por: Gustavo A. Galeano A.
#define __FREESCALE_
#ifndef __FREESCALE_
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#endif
#include "serial.h"
void Serial_Init(void){
#ifndef __FREESCALE_
    //Función de Inicialización del Serial a 9600 8N1
    /* SCC1: LOOPS=0,ENSCI=0,TXINV=0,M=0,WAKE=0,ILTY=0,PEN=0,PTY=0 */
    SCC1 = 0x00; /* Configure the SCI */
    /* SCBR: ??=0,?=0,SCP1=0,SCP0=0,?=0,SCR2=0,SCR1=1,SCR0=0 */
    SCBR = 0x02; /* Set the selected baud rate */
    /* SCC2: SCTIE=0,TCIE=0,SCRIE=0,ILIE=0,TE=0,RE=0,RWU=0,SBK=0 */
    SCC2 = 0x00; /* Disable all interrupts */
    SCC1_ENSCI = 1; /* Enable device */
    /* SCC3: ORIE=1,NEIE=1,FEIE=1,PEIE=1 */
    SCC3 |= 0x0F; /* Enable error interrupts */
    SCC2 |= ( SCC2_TE_MASK | SCC2_RE_MASK); /* Enable Tx y Rx*/
#endif
}
void Serial_SendChar(char Chr){ /*Envío de caracter vía serial*/
#ifndef __FREESCALE_
    while(!SCS1_SCTE); /*espera que esté vacio*/
    SCDR = Chr; /*envía dato al periférico SCI*/
    while(!SCS1_TC); /*espera que se transmita el dato*/

```



```
#endif
#ifndef __PIC__
    putc(Chr);
#endif
}

#ifndef __FREESCALE__
void Serial_SendMsg(char* Ptr){ /*Envía cadena de datos vía serial*/
static char *ptrData;
ptrData = Ptr;
while(*ptrData){
    Serial_SendChar(*ptrData); /* Envía el Dato */
    ptrData++;
}
#endif
#ifndef __PIC__
#define Serial_SendMsg printf
#endif
char GetChar(void){
#ifndef __FREESCALE__
    while(!SCS1_SCRF); /*Espera que llegue un carácter*/
    return SCDR;
#endif
#ifndef __PIC__
    return getch();
#endif
}
/*Función de recepción de comandos en minúscula*/
void Serial_RecibirCmd(char *bufferRxPtr){
unsigned char i=0;

do{
    bufferRxPtr[i] = GetChar(); //lo almacena en el
    buffer de Rx
    Serial_SendChar(bufferRxPtr[i]); //ECO de la
    recepción
    i++;
}while((bufferRxPtr[i-1]!= ENTER) &&
(i < (BUF_SIZE-1)));
bufferRxPtr[i-1] = 0;
}
```

7.6 LIBRERÍA DE TIPOS <CTYPE.H>



Con las funciones de la librería <ctype.h> el programador recibe un FALSE o TRUE dependiendo de si un carácter pertenece o no a un conjunto de datos.



La librería ctype contiene un grupo de funciones implementadas que determinan si un carácter pertenece a un conjunto específico de datos, retornando FALSE o TRUE si el carácter que se entrega como argumento está, por ejemplo, dentro de un conjunto numérico (**isalnum**), alfabético (**isalpha**), carácter de control (**iscntrl**), si es una letra mayúscula o minúscula (**isupper**, **islower**).

Complementa el grupo un par de funciones **tolower**, **toupper**, que convierten el carácter alfabético entregado como argumento a su

respectivo valor minúscula o mayúscula respectivamente.

Materiales adicionales en la



Lista de funciones de la librería <ctype.h> en el Anexo # 2, y explicación de cada función.

EJEMPLO No. 30

Manejo de cadenas

Objetivo

Usando el puerto serial del microcontrolador, diseñar software en C para el manejo de una estructura de datos de un grupo de usuarios que contiene los siguientes campos:

nombre: cadena de caracteres ASCII.

email: cadena de caracteres con email válido.

edad: un carácter no signado que indica la edad en años del usuario.

La interfaz serial deberá permitir los siguientes comandos y acciones:

>prom Permite consultar el promedio de edades de los usuarios.

>nombres Envía los nombres, email y edad de los usuarios.

>maxname Envía el nombre del usuario de mayor longitud de caracteres.

>nombre i y modifica el nombre i por el nuevo nombre y.

>email j z modifica el email j de la estructura por el nuevo email z.

>edad k x Modifica la edad del componente k de la estructura por x.

>chkemails Envía, vía serial, los emails inválidos.

Solución:

La solución se apoya en un nuevo módulo para el manejo del interfaz SCI del microcontrolador que se llamará SERIAL, las funciones públicas implementadas e incluidas en el archivo SERIAL.H son:

```

/* SERIAL.H *
// Módulo de Manejo Serial
#ifndef __SERIAL_H
#define __SERIAL_H
#define ENTER 0x0D //definición de constante retorno
#define LINE_FEED 0x0A //definición de constante avance
#define BUF_SIZE 21
void Serial_Init(void); //Función de Inicialización del Serial a 9600 8N1
void Serial_SendChar(char Chr); // Función de envío de un carácter vía serial
void Serial_SendMsg(char* Ptr); // Función de envío de una cadena de datos
vía serial
void Serial_RecibirCmd(char *bufferRxPtr); // Función de recepción de datos
vía serial SCI1
char GetCh(void); // Función de recepción de un carácter serial
#endif

/* SERIAL.C *
// Código fuente SERIAL.C
// Asunto: Módulo de Manejo del SCI AP16 9600b 8N1
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "serial.h"
//Función de Inicialización del Serial a 9600 8N1
void Serial_Init(void){
    /* SCC1: LOOPS=0,ENSCI=0,TXINV=0,M=0,WAKE=0,ILTY=0,PEN=0,PTY=0 */
    SCC1 = 0x00; /* Configure the SCI */
    /* SCBR: ??=0,?=0,SCP1=0,SCP0=0,?=0,SCR2=0,SCR1=1,SCR0=0 */
    SCBR = 0x02; /* Set the selected baud rate */
    /* SCC2: SCTIE=0,TCIE=0,SCRIE=0,ILIE=0,TE=0,RE=0,RWU=0,SBK=0 */
    SCC2 = 0x00; /* Disable all interrupts */
    SCC1_ENSCI = 1; /* Enable device */
    /* SCC3: ORIE=1,NEIE=1,FEIE=1,PEIE=1 */
    SCC3 |= 0x0F; /* Enable error interrupts */
    SCC2 |= ( SCC2_TE_MASK | SCC2_RE_MASK); /* Enable transmitter, Enable Receiver*/
}
void Serial_SendChar(byte Chr){ // Función de envío de un carácter vía serial
    while(!SCS1_SCTE); //espera que este vacío
    SCDR = Chr; //envía dato al periférico SCI
    while(!SCS1_TC); //espera que se transmita el dato
}
void Serial_SendMsg(byte* Ptr){ // Función de envío de una cadena de datos vía Serial
static byte *ptrData;
    ptrData = Ptr;
    while(*ptrData){
        Serial_SendChar(*ptrData); /* Envía el Dato */
        ptrData++;
    }
}

```

```

char GetCh(void){ // Función de recepción de carácter serial
    while(!SCS1_SCRF); // Espera que llegue un carácter
    return SCDR; // lo almacena en el buffer de Rx
}
void Serial_RecibirCmd(char *bufferRxPtr){ // comandos en letras minúscula
unsigned char i=0;
do{
    while(!SCS1_SCRF); // Espera que llegue un carácter
    bufferRxPtr[i] = SCDR; // lo almacena en el buffer de Rx
    Serial_SendChar(bufferRxPtr[i]); // ECO de la recepción
    i++;
}while((bufferRxPtr[i-1]!=ENTER) && (i < (BUF_SIZE-1)));
bufferRxPtr[i-1] = 0;
}
//***** Ejemplo 30 *****
// Manejo de Cadenas
// Fecha: Abril 11,2009
// Asunto: Manejo de Interfaz Serial con Comandos
// Hardware: Sistema de desarrollo AP-Link
// FreescaleTM.
// Version: 1.0 Por: Gustavo A. Galeano A.

//*****
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include <stdlib.h> // para el uso de atoi()
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "serial.h"
void FxProm(void); /* prototipos */
void FxNombres(void);
void FxChkEmails(void);
void FxMaxName(void);
void FxNewNameEmail(void);
void FxNewEdad(void);
void FxHelp(void);
#define CMD_PROM "PROM" /* definición comandos */
#define CMD_NOMBRES "NOMBRES"
#define CMD_CHKEMAILS "CHKEMAILS"
#define CMD_MAXNAME "MAXNAME"
#define CMD_NEW_NAME "NOMBRE"
#define CMD_NEW_EMAIL "EMAIL"
#define CMD_EDAD "EDAD"
#define CMD_HELP "?"
#define H_PROM "PROM<enter> --> Promedio Edades\r\n"
#define H_NOMBRES "NOMBRES<enter> --> Lista de Nombres\r\n"
#define H_CHKEMAILS "CHKEMAILS<enter> --> Lista de Emails\r\n"

```

```

#define H_MAXNAME "MAXNAME<enter> --> Nombre de Mas Caracteres\r\n"
#define H_NEW_NAME "NOMBRE nroUsuario NuevoNombre<enter> --> Cambio Nombre\r\n"
#define H_NEW_EMAIL "EMAIL nroUsuario NuevoEmail<enter> --> Cambio de Email\r\n"
#define H_EDAD "EDAD nroUsuario NuevaEdad --> Cambio de Edad\r\n"
#define H_HELP "?<enter> --> Ayuda de Comandos\r\n"
typedef void vFuncPtrV(void);
#define NRO_MAX_CH_EMAIL 25
struct Usuario{ /*Definición de la Estructura Usuario*/
    char user[15];
    char password[7];
    char email[NRO_MAX_CH_EMAIL];
    unsigned char edad;
};
struct ComandosStr{ /*Definición estructura Comandos*/
    char *comandNamePtr;
    char *comHelp;
    vFuncPtrV *FxPtr;
};
/*Variables Globales*/
//Almacenamiento de la estructura en RAM con datos de
//Inicialización de Flash
struct Usuario usuarios[]={
    "Alberto","clave1","albert@alfaomega.com.mx",23,
    "Beatriz","clave2","betty@hotmail.com",24,
    "Camilo","clave3","camilo@myemail.com",28,
    "Daniela","clave4","dany@mycity.com",18,
    "Kimberly","clave5","kim@user.com",35,
    0,0,0,0,
};
// Almacenamiento de los posibles comandos en FLASH
const struct ComandosStr comandosStr[]={
    CMD_PROM, H_PROM, FxProm,
    CMD_NOMBRES, H_NOMBRES, FxNombres,
    CMD_CHKEMAILS,H_CHKEMAILS,FxChkEmails,
    CMD_MAXNAME, H_MAXNAME, FxMaxName,
    CMD_NEW_NAME, H_NEW_NAME, FxNewNameEmail,
    CMD_NEW_EMAIL,H_NEW_EMAIL,FxNewNameEmail,
    CMD_EDAD, H_EDAD, FxNewEdad,
    CMD_HELP, H_HELP, FxHelp,
    NULL, NULL, NULL,
};
#define NRO_MAX_BUFFER 40 /*Buffer de recepción Serial*/
char bufferRx[NRO_MAX_BUFFER];
ulong promedioEdades;
/*FUNCIONES*/
void Mcu_Init(void){ /*Inicialización de la CPU*/
/*
COPRS=0,LVISTOP=0,LVIRSTD=0,LVIPWRD=0,LVIREGD=0,SSREC=0,STOP=0, COP

```

CONFIG1:

```

D=1 */
CONFIG1 = 0x01;
/*
STOP_ICLKDIS=0,STOP_RCLKEN=0,STOP_XCLKEN=0,OSCCLK1=0,OSCCLK0 =0, ???
=0,??=0,SCIBDSRC=1 */
CONFIG2 = 0x01;
}
void Serial_RxCommand(void){/*Recepción serial de datos*/
uchar i=0;
uint longBuffer;
char *bufferPtr;
Serial_SendMsg("\r\n\ncomando?>");
do{
    bufferRx[i] = GetCh();
    Serial_SendChar(bufferRx[i]); //Eco de recepción
    i++;
}while((bufferRx[i-1]!= ENTER) && (i < NRO_MAX_BUFFER));
bufferRx[i-1] = 0;
longBuffer = strlen(bufferRx);
bufferPtr = memchr(bufferRx, ' ',NRO_MAX_BUFFER);
if(bufferPtr != NULL){
    *bufferPtr = '\0'; //el primer espacio lo cambia por NULL
    bufferPtr++;
    bufferPtr = memchr(bufferPtr, ' ',(NRO_MAX_BUFFER-longBuffer));
    if(bufferPtr != NULL){
        *bufferPtr = '\0'; //segundo espacio
    }
}
}
void FxProm(void){/*Cálculo del promedio de edades*/
uchar i;
promedioEdades = 0;
for(i=0;usuarios[i].edad;i++){
    promedioEdades += usuarios[i].edad;
}
promedioEdades = promedioEdades/(ulong)i;
(void)sprintf(bufferRx,"Promedio: %f",promedioEdades);
Serial_SendMsg(bufferRx);
}
// Función de envío de todos los nombres
// en la estructura usuarios
void FxNombres(void){
uchar i;
for(i=0;usuarios[i].edad;i++){
    Serial_SendMsg(usuarios[i].user);
    Serial_SendMsg("\r\n");
}
}

```

```

char UserOk(char *userPtr){/*Verificación de usuario de email*/
char *ptrTmp;
uchar i,lngUser;
    lngUser = (uchar)strlen(userPtr);
    ptrTmp = strchr(userPtr,'@');
    if(ptrTmp == NULL){ //ok, no tiene '@'
        for(i=0;i<lngUser;i++){
            if(iscntrl(userPtr[i]) || isspace(userPtr[i])) return 0;
        }
        return 1;
    }
    return 0;
}
/*Función de verificación de los caracteres del servidor*/
char ServerOk(char *serverPtr){
char *ptrTmp;
uchar i,lngUser;
    lngUser = (uchar)strlen(serverPtr);
    ptrTmp = strchr(serverPtr,'@');
    if(ptrTmp == NULL){ //ok, no hay mas '@'
        for(i=0;i<lngUser;i++){
            if(iscntrl(serverPtr[i]) || isspace(serverPtr[i])) return 0;
        }
        ptrTmp = strchr(serverPtr,'.');
        if(ptrTmp){ //existe un '.' al menos
            return 1;
        }
    }
    return 0;
}
/*Función de chequeo de un email, retorna TRUE si está bien
o FALSE si está mal conformado*/
char EmailOk(char *emailPtr){
char emailBck[NRO_MAX_CH_EMAIL];
char *ptrTmp,i;
    for(i=0;i<NRO_MAX_CH_EMAIL;i++) emailBck[i] = emailPtr[i];
    ptrTmp = strchr(emailBck,'@');
    if(ptrTmp){ // ok, hay mínimo un '@'
        ptrTmp[0] = '\0';
        ptrTmp++;
        if(UserOk(emailBck)){
            if(ServerOk(ptrTmp)){ return 1; }
        }
    }
    return 0;
}
// Función de chequeo correcto de emails
void FxChkEmails(void){

```

```

uchar i;
for(i=0;usuarios[i].email;i++){
if(!EmailOk((char *)&usuarios[i].email)){
    Serial_SendMsg("Error Email: ");
}else{
    Serial_SendMsg("Email OK: ");
}
Serial_SendMsg((unsigned char *)&usuarios[i].email);
}

void FxMaxName(void){
uchar maxNameID=0,i;
uint nroMax=0,nroMaxTemp;
for(i=0;usuarios[i].edad;i++){
    nroMaxTemp = strlen(usuarios[i].user);
    if(nroMax < nroMaxTemp){
        maxNameID = i;
        nroMax = nroMaxTemp;
    }
}
Serial_SendMsg(usuarios[maxNameID].user);
}

// Función de Ingreso de cambio de nombre o email
void FxNewNameEmail(void){
int offset;
static char *newPtr=bufferRx; // static para que sea más rápido
newPtr += strlen(bufferRx);
newPtr++;
offset = atoi(newPtr)-1;
newPtr += strlen(newPtr);
newPtr++;
if(!strcmp(bufferRx,CMD_NEW_NAME)){ //es cambió de nombre?
    (void)strcpy(&(usuarios[offset].user),newPtr);
}else{
    if(!strcmp(bufferRx,CMD_NEW_EMAIL)){ //es cambió de email?
        (void)strcpy(&(usuarios[offset].email),newPtr);
    }
}
}

void FxNewEdad(void){ /*Función de cambio de edad*/
int offset;
static char *newPtr=bufferRx; /*static más rápido*/
newPtr += strlen(bufferRx);
newPtr++;
offset = atoi(newPtr)-1;
newPtr += strlen(newPtr);
newPtr++;
usuarios[offset].edad = (unsigned char)atoi(newPtr);
}

```

```

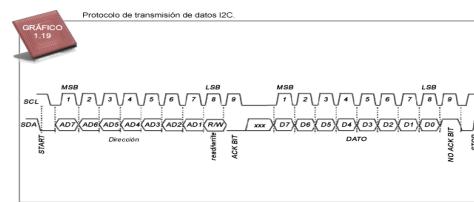
}

void FxHelp(void){ /*Función de Solicitud de ayuda*/
uchar i;
for(i=0;comandosStr[i].comHelp!= NULL;i++){
    Serial_SendMsg(comandosStr[i].comHelp);
}
}

void Serial_Exec(void){
uchar i=0,commandFound=0;
Serial_SendMsg("\r\n\n");
do{
if(!strcmp(bufferRx,comandosStr[i].comandNamePtr)){
    comandosStr[i].FxPtr();
    commandFound = 1;
}
i++;
}while((comandosStr[i].comandNamePtr != NULL) && (!commandFound));
}

void main(void){ /*Función principal*/
Mcu_Init();
Serial_Init();
EnableInterrupts; /* enable interrupts */
Serial_SendMsg("\r\n*** ALFAOMEGA GRUPO EDITOR ****\r\n");
Serial_SendMsg("\r\n www.alfaomega.com.mx\r\n");
Serial_SendMsg("\r\n-- Ejemplo Manejo de Cadenas --\r\n");
for(;;){
    Serial_RxCommand();
    Serial_Exec();
} /* loop forever */
}

```



Discusion:

La implementación de la solución está apoyada en el módulo SERIAL que contiene las funciones de bajo nivel de configuración, envío y recepción de caracteres vía serial SCI.

El módulo principal inicializa los módulos y permanece en un ciclo sin fin, que recibe un comando vía serial, una vez recibido, invoca la función Serial_Exec() que resuelve el comando recibido, esta función entra en un nuevo ciclo do{}while, que busca el comando e invoca a su vez la función correspondiente.

La función FxChkEmails() está basada en lo siguiente:

Un email debe tener un y solo un carácter '@'.

Los caracteres que anteceden el carácter '@', no puede contener caracteres de control, o espacios.

La función FxProm() hace uso de función sprintf() que toma el valor de la variable promedioEdades y la convierte en formato de cadena en bufferRx. La interfaz hyperterminal permitirá al usuario entrar los comandos y visualizar los resultados en la pantalla del PC.



```
----- ALFAOMEGA GRUPO EDITOR -----
www.alfaomega.com.mx
----- Ejemplo Manejo de Cadenas -----
comando?>?
PROM<enter> --> Promedio Edades
NOMBRES<enter> --> Lista de Nombres
CHKEMAILS<enter> --> Lista de Emails
MAXNAME<enter> --> Nombre de Mas Caracteres
NOMBRE nroUsuario NuevoNombre<enter> --> Cambio Nombre
EMAIL nroUsuario NuevoEmail<enter> --> Cambio de Email
EDAD nroUsuario NuevaEdad --> Cambio de Edad
?<enter> --> Ayuda de Comandos

comando?>

comando?>_
```

2:03:42 conectado Autodetect. 9600 B-N-1 DESPLAZAR MAY NUM Capturar Imprimir

7.7 LIBRERÍA DE MANEJO DE TIEMPO <TIME.H>



Las funciones disponibles en la librería <time.h> posibilitan la programación de eventos asociados al manejo del tiempo, sistemas de conteo, y

Materiales adicionales en la



[Lista de funciones de la librería <time.h> en el Anexo # 2, y explicación de cada función.](#)

 Las funciones de la librería time son útiles para el conteo, conversión y manejo del tiempo de la vida real, denominado **RTC** (*Real Time Clock*).

El tiempo, tal como se conoce en la vida real, está representado por el ANSI en una estructura denominada **time_t**, que contiene en diferentes campos, cada uno de sus características como: número de segundos, número de minutos, hora actual, día del mes, número del mes y otros, que describen el segundo actual como real y único.

Cada segundo, la estructura cambia, dado por el cambio del campo de segundos, sin embargo, el manejo de la estructura en cada segundo resultaría muy ineficiente tanto en tiempo de ejecución como en consumo de energía, porque en cada segundo se necesitaría examinar si los campos cambian, así cada que el campo de segundos llegara a 59, debería pasar a 0 (cero), incrementar el campo de minutos, y si este a su vez llega a 59, también pasar a 0(cero), y así sucesivamente, verificar todos los campos.

La estructura **tm** del ANSI C es definida en <time.h> mínimo con los siguientes campos:

```
struct tm {
    > int tm_sec; /segundos después del minuto --- [0-59]
    > int tm_min; /minutos después de hora ----- [0-59]
    > int tm_hour; /horas desde la medianoche ---- [0-23]
    > int tm_mday;/día del mes ----- [1-31]
    > int tm_mon; /mes desde Enero ----- [0-11]
    > int tm_year; /años desde 1900 ----- [108 es 2008]
    > int tm_wday;/día semana desde Domingo ----- [0-6]
    > int tm_yday;/días desde Enero ----- [0-365]
    > int tm_isdst; /bandera del ahorro de energía -- [0 ó 1]
};<p>
```

El valor de **tm_isdst** (**is daylight saving**), es positivo si el horario de ahorro de energía está en efecto, cero si no lo está y negativo si la información no está disponible.

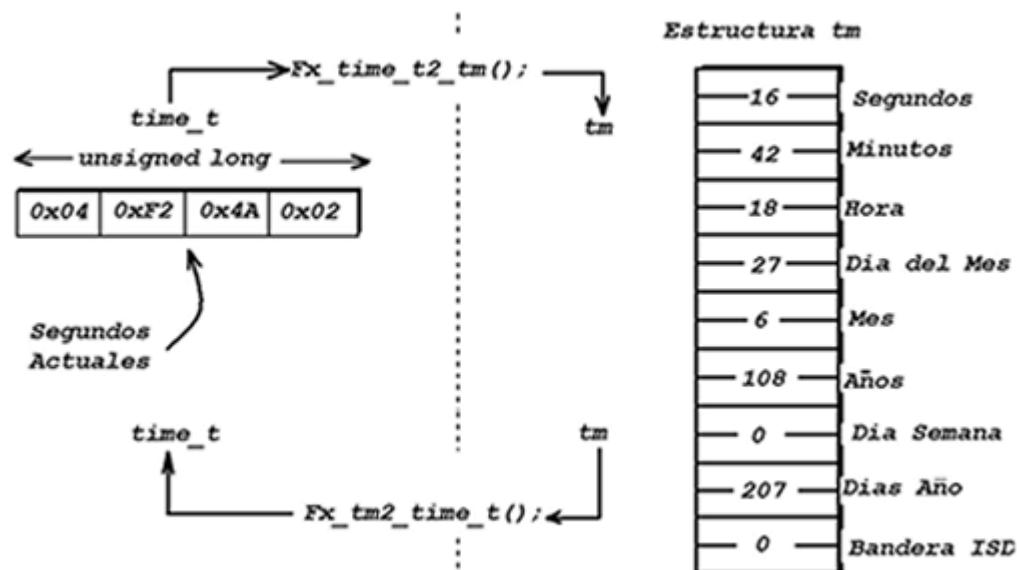
La forma sencilla y eficiente de llevar el tiempo en procesadores, consiste en contabilizarlo de forma aritmética con una variable **unsigned long (4 bytes)** que está definida por <time.h> como **time_t**.

Esta variable se inicializa con el número de segundos transcurrido desde el 1 de Enero de 1970.

Esta representación no tiene sentido para el usuario del mundo real, lo mismo que el valor de inicialización es indeterminado, por esta razón se usan funciones de conversión entre **tm** y **time_t**, y entre **time_t** y **tm**, dependiendo si se está haciendo el manejo en la máquina (**time_t**) o en la representación real (**struct tm**) (ver Gráfico 7.10).



Representación del tiempo en forma binaria y real.



RESUMEN DEL CAPÍTULO

Las librerías del ANSI C proporcionan un grupo de Funciones a las que el programador puede acudir para resolver alguna operación compleja, permite el ahorro de tiempo de implementación y prueba de rutinas. Debe prestarse especial atención a la forma como los argumentos a las funciones de una librería son entregadas a fin de obtener los resultados de forma correcta, estos argumentos deben corresponder a los publicados en los archivos de cabecera (.H).

La librería `<math.h>` pone a disposición operaciones complejas de tipo flotante, por su parte la librería `<stdlib.h>` además de algunas conversiones útiles de caracteres, permiten el manejo óptimo de memoria usando la localización dinámica.

Usar la librería `<string.h>` al requerir realizar tratamiento con cadenas de caracteres, mover, concatenar y operaciones de búsqueda.

El manejo del tiempo usando la librería `<time.h>`, facilita por medio de funciones de cambio entre las variables de manejo de conteo digital (`time_t`) y la representación por campos (`struct tm`) para mostrar el tiempo tal como se conoce en la vida cotidiana.

El programador deberá medir el impacto en tiempo de ejecución y el tamaño del código generado al incluir una librería o alguna de sus funciones, en algunos casos resulta más eficiente realizar rutinas específicas que realicen el procedimiento particular y no incluir todas las funcionalidades de una rutina.

PREGUNTAS Y EJERCICIOS PROPUESTOS

Implementar el ejemplo de la calculadora serial, usando un display LCD para la visualización de datos y un teclado matricial.

¿Qué limitaciones tiene el uso de la librería **<math.h>** sobre una tabla de datos para resolver una operación matemática? Realizar una función específica para resolver la función seno de un ángulo dado en radianes, usando una tabla y medir tiempos de ejecución y memoria utilizada y compare.

¿En qué fecha futura la variable **unsigned long time_t** tendrá sobre flujo? ¿Qué implicaciones puede tener en un sistema embebido?

Realizar un código que vía serial reciba una fecha pasada del día de cumpleaños de una persona y muestre vía serial el día de la semana que esa persona nació: Lunes, Martes, Miércoles, Jueves, Viernes, Sábado o Domingo.

INTRODUCCIÓN

El control del consumo de energía es un aspecto cada vez más importante a considerar en los diseños embebidos dada su creciente proliferación.

En este capítulo se muestran algunas técnicas útiles que permiten a un equipo funcionar por más tiempo con una fuente de energía externa, se muestran las ventajas en espacio y control de temperatura final del sistema.

Este tema muestra el porqué los fabricantes de semiconductores están invirtiendo cada vez más en investigaciones sobre este aspecto, limitados a reducir el consumo de energía por las exigencias del mundo actual, las cuales tienden a ser, con el paso de los días, aún más drásticas.

Sin embargo, aunque la tecnología sea de bajo consumo, el software embebido que va dentro de cada procesador es el que finalmente puede administrar este factor de forma óptima y corresponde al programador limitar el uso, desperdicio o ahorro. De allí la importancia de adoptar las técnicas descritas acá en la programación rutinaria del diseñador.

Se muestra de forma cualitativa los efectos de dos de los modos más populares de consumo: el WAIT y el STOP, pero se hace referencia a otros modos de familias más modernas como lo es la HCS08 Freescale™ con la incorporación de los modos Stop3, Stop2 y Stop1 y la familia Flexis™ de Freescale™ que ingresa los modos LPrun y LPwait.

Al final son listadas algunas consideraciones generales que ayudan a que el consumo sea menor en el sistema embebido.

8.1 LA ECUACIÓN DE CONSUMO DE ENERGÍA

Un sistema digital está operando continuamente cambiando bits de estado uno a cero y de cero a uno, estos cambios originan inestabilidad momentánea no deseada, denominada transiente, que alteran el consumo de energía en los momentos de transición.

El consumo de potencia debido a transientes de AC y DC se incrementan con la frecuencia de operación y el voltaje de alimentación. La potencia de AC la describe la expresión CV^2f , que corresponde a la potencia disipada al manejar una capacitancia de carga C , incrementa proporcionalmente con la frecuencia de operación f y con el cuadrado del voltaje de alimentación V , obviamente con la capacitancia de carga, pero está dada por el sistema y no se puede alterar.



El consumo de energía del microcontrolador está regido por la frecuencia de operación y el voltaje de alimentación.



La potencia de DC por su lado está dada por la multiplicación de voltaje **V** y Corriente **I** durante las transiciones de las compuertas.

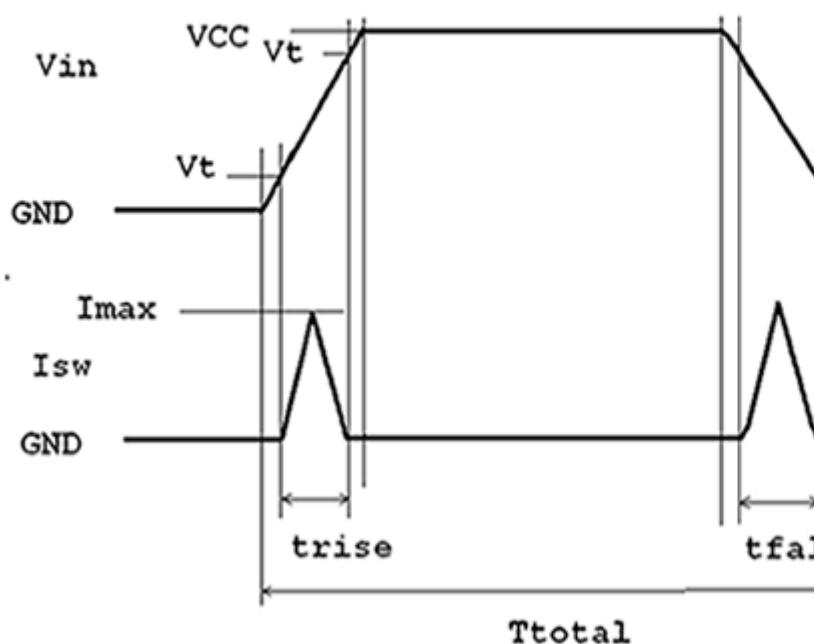
En cualquier dispositivo CMOS durante las transiciones existe un pequeño instante en el que la corriente circula desde VCC a GND.

La magnitud de la potencia disipada de DC está determinada por tres factores: el voltaje de alimentación VCC, la frecuencia de operación **f** y el tiempo de transiciones **trise** (Rise Time: trise) (*ver Gráfico 8.1*).



Potencia disipada en transiciones.

GRÁFICO
8.1



Existen dos modos básicos para mantener el procesador HC08

en estado de bajo consumo: Wait y Stop.



Potencia disipada en transiciones.

$$PVI = VCC * \frac{1}{2} * Imax * ((VCC - 2Vt) / VCC) * ((trise + tfall) / Ttotal)$$

Donde $1/Ttotal$ es la frecuencia f

Queda:

$$PVI = \frac{1}{2} (VCC - 2Vt) * Imax * (trise + tfall) * f$$

Se puede notar que si el tiempo de transición (**trise**), es muy largo, la potencia disipada se incrementa, dado que **Imax** permanece por más tiempo; en teoría, si el tiempo de transición es nulo, la potencia dada por VI sería cero; sin embargo, en los circuitos de la vida real no es posible obtener dicho valor y es allí donde el voltaje de alimentación hace su aparición en la ecuación final.

Tener varios modos de funcionamiento en un procesador, permite administrar el consumo de energía de forma eficiente, entregar la medida suficiente que se requiere para los diferentes estados de funcionamiento y optimizar el uso de las baterías o fuentes que soporta el sistema embebido.

De forma general existen en el procesador dos tipos de bajo consumo: **Wait** y **Stop**, los cuales son invocados dependiendo de los eventos requeridos para despertar al procesador de el estado de bajo consumo.



Para minimizar el consumo de potencia de un sistema, se deberá ejecutar a la mínima velocidad y el menor voltaje de alimentación posible.

8.2 MODO "WAIT"



El modo Wait ahorra un 50% de energía al deshabilitar el reloj de la CPU y dirigir el

A este modo se entra mediante la ejecución de la instrucción inherente **WAIT**. En este modo, el reloj que mueve la CPU es deshabilitado, evitando que siga la ejecución; sin embargo, el reloj continúa a su frecuencia normal de trabajo alimentando los diversos periféricos que lo requieren, por ejemplo, el convertidor AD, comunicaciones SCI, SPI, I2C, temporizadores y demás periféricos que requieran de la frecuencia del reloj si se está en **WAIT**.

consumo únicamente a los periféricos que lo requieren.



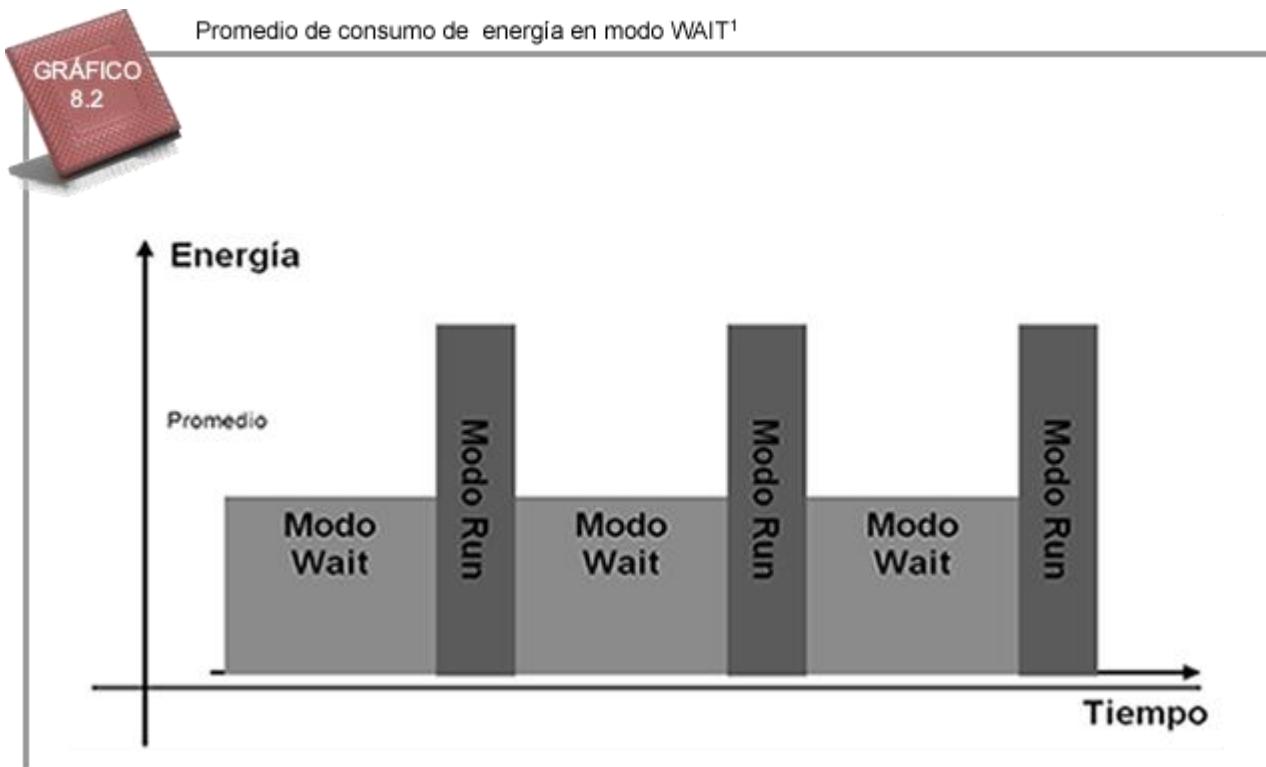
El consumo de corriente en este modo se reduce aproximadamente a un 50% de la corriente nominal de ejecución (modo **RUN**), ante el mismo voltaje de alimentación. De este modo de bajo consumo el procesador puede salir por las siguientes razones:

Reset: por cualquiera de sus causas.

Interrupción externa: dada por el cambio de un pin externo que la genere.

Interrupción interna: cualquiera de los periféricos que genere una solicitud a la CPU.

Al entrar al modo **WAIT** el procesador borra la bandera de interrupción I del **CCR**, para permitir así que pueda ser sacado de este modo al presentarse una interrupción.



8.3 MODO "STOP"



El modo Stop ahorra más de un 90% de consumo de energía, y consiste en la detención del microcontrolador, pero conservando ciertas interrupciones que lo pueden sacar de ese estado.



A este modo se entra ejecutando la instrucción inherente **STOP**, la cual elimina toda la disipación dinámica de potencia, parando por completo el oscilador principal del procesador.

Antes de entrar a este modo el bit de **I** (*interrupt*) dentro del CCR es borrado, para permitir que ciertas interrupciones lo saquen de este modo.

Esta instrucción, como se puede notar, parará la ejecución del microcontrolador, por lo que su ejecución deberá realizarse con precaución; una ejecución indebida o no deseada de esta instrucción hará que el microcontrolador quede estático y no continúe con la ejecución normal del programa y en caso que no se programen las interrupciones debidas que lo saquen de este estado, solo un *Reset* podría recuperar el sistema.

Para prevenir que el procesador entre a este modo de forma accidental, se dispone de un bit en uno de los bits de configuración denominado **STOPE** (*Stop Enable*) que habilita la ejecución de esta instrucción. Si este bit está apagado, la instrucción STOP no se ejecutará y será tratada como una instrucción ilegal, generando la interrupción respectiva.



La miniaturización de los equipos implica que los Sistemas Embebidos deben presentar un gran ahorro energético, lo que se traduce en la exigencia al programador para desarrollar un código que aplique eficientemente el consumo de energía.

En este modo el consumo del procesador se reduce a niveles muy bajos, del orden de los 3 μ A.

De este modo el procesador puede salir por las siguientes razones:

Reset: por cualquiera de sus causas.

Interrupción externa: IRQ o KBI.

Interrupción interna TBM: temporizador de Base de Tiempo, solo si el bus está habilitado para este módulo.

En las familias de microcontroladores de Freescale™ **HCS08** existen a su vez tres modos de Stop denominados **Stop1**, **Stop2** y **Stop3**, los cuales apagan algunos periféricos internos permitiendo que otros sigan activos, y de esta forma tener un consumo optimizado dependiendo de las funciones que requieran estar encendidas sin permanecer en el modo Wait. A cualquiera de los modos se entra mediante la instrucción STOP; sin embargo, es mediante algunos bits que se configura el modo en particular.

8.3.1 Modo “Stop3”

Una vez se entra al modo **stop3**, todos los relojes dentro del microcontrolador, incluido el mismo oscilador son parados. El reloj interno entra en modo de *standby*, y lo mismo lo hace el regulador de voltaje y el convertidor analógico a digital. El estado de todos los registros de la RAM lo mismo que el de los registros internos son mantenidos, sosteniendo en virtud de ello el estado de los pines de entrada/salida (*ver Gráfico 8.3*).

La salida de modo **Stop3** se realiza por medio de un *Reset*, una interrupción asincrónica IRQ, KBI, LVD, o por el **RTI** (*Real Time Interrupt*).

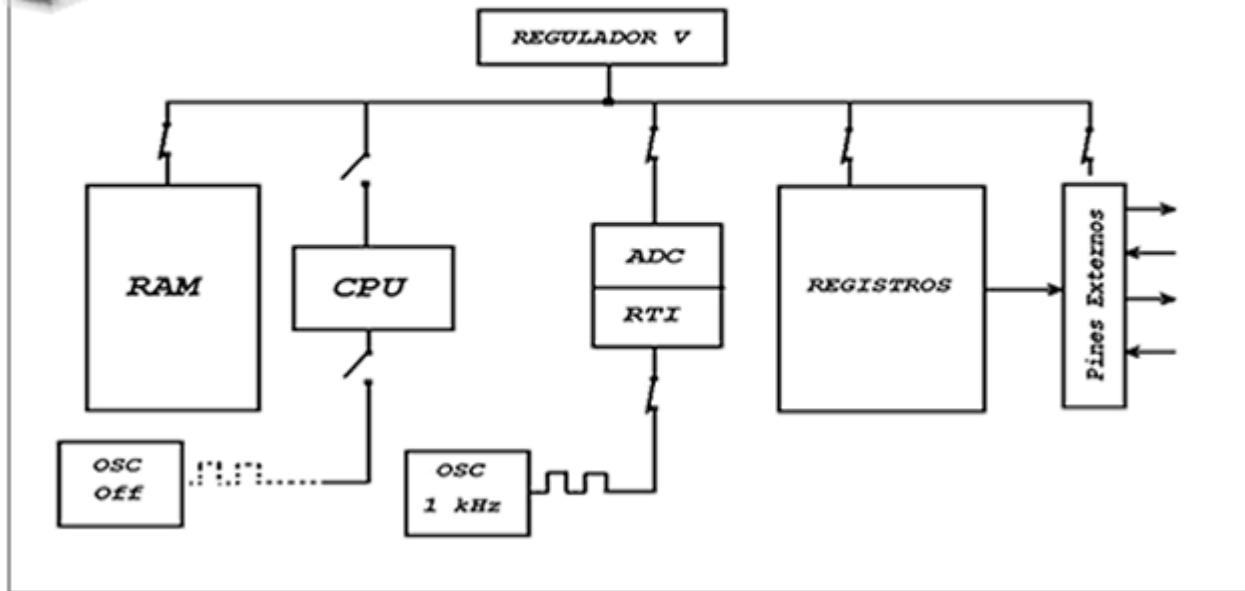
Si la salida es por medio del pin *Reset*, el **MCU** ejecutará la secuencia de Reset tomando el vector de Reset, mientras que si se sale de **Stop3** por una interrupción asincrónica: IRQ, KBI o RTI, el MCU tomará el respectivo vector de interrupción y una vez procesada la ISR, el flujo del programa retornará a la instrucción que sigue la instrucción **STOP**.

Una fuente separada de reloj (1 KHz), para el RTI permite salir de los modos de bajo consumo **Stop2** o **Stop3**.

El límite de potencia para los procesadores de última generación Flexis está en los **450nA**.



Modo Stop3 de bajo consumo.



8.3.2 Modo “Stop2”



El modo Stop2 prioriza el consumo a la RAM, donde se encuentran almacenados los registros de entrada/salida y cualquier otro que pueda ser restablecido; el resto de circuitos internos son apagados.

El modo Stop2 provee un modo de muy bajo consumo manteniendo además los contenidos de la memoria **RAM** y el estado actual de los pines de entrada/salida. Para seleccionar la entrada al modo **Stop2** además de la ejecución de la instrucción **STOP**, deberán ponerse en 1 los bits **PDC** y **PPDC** en el registro **SPMSC2**.

Antes de entrar a este modo, el programa debe guardar los contenidos de los registros de entrada/salida, así como cualquier otro registro que desee ser restablecido una vez se sale del modo **Stop2**. Una vez entrado en modo **Stop2**, todos los circuitos internos que son alimentados por el regulador de voltaje son apagados, excepto el que alimenta la memoria **RAM**. El regulador de voltaje estará en modo de bajo consumo *standby* así como el convertidor AD. Una vez se entra al modo de bajo consumo, los estados de los pines de entrada/salida son mantenidos mientras se está en el modo **Stop2** y después de salir del modo **Stop2** hasta que un 1 lógico es escrito en el bit **PPDACK** en el registro **SPMSC2**.

La salida del modo **Stop2** se realiza por medio de tres formas: **Reset**, **IRQ** o una interrupción **RTI** (**Real Time Interrupt**); la interrupción **IRQ** deberá ser siempre **activa en bajo** cuando el MCU esté en modo **Stop2**, independiente de cómo fue configurada antes de entrar al modo **Stop2**.

Una vez se sale del modo **Stop2**, el **MCU** arrancará como lo hace de **POR (Power On Reset)**, excepto que el estado de los pines permanece sostenido. La **CPU** tomara el vector de Reset y el sistema y todos los periféricos estarán en su estado inicial de Reset y deberán ser inicializados.

Para mantener el estado intacto de los pines de entrada/ salida, el usuario deberá restablecer el contenido de los registros correspondientes a los puertos de entrada/salida, que debieron ser almacenados en **RAM**, a los registros antes de escribir al bit **PPDACK**. Si los valores de los registros no son almacenados antes de escribir al bit **PPDACK**, los registros asumirán su valor de Reset, y los pines de entrada/salida tomaran esos valores.

Para los pines que fueron configurados como periféricos, el usuario deberá reconfigurarlos de nuevo antes de escribir el bit **PPDACK**. Si el módulo periférico no es habilitado antes de escribir el bit **PPDACK**, los pines serán controlados por su registro de control de puerto correspondiente.

El límite de potencia para los procesadores de última generación Flexis está en los 370 nA.

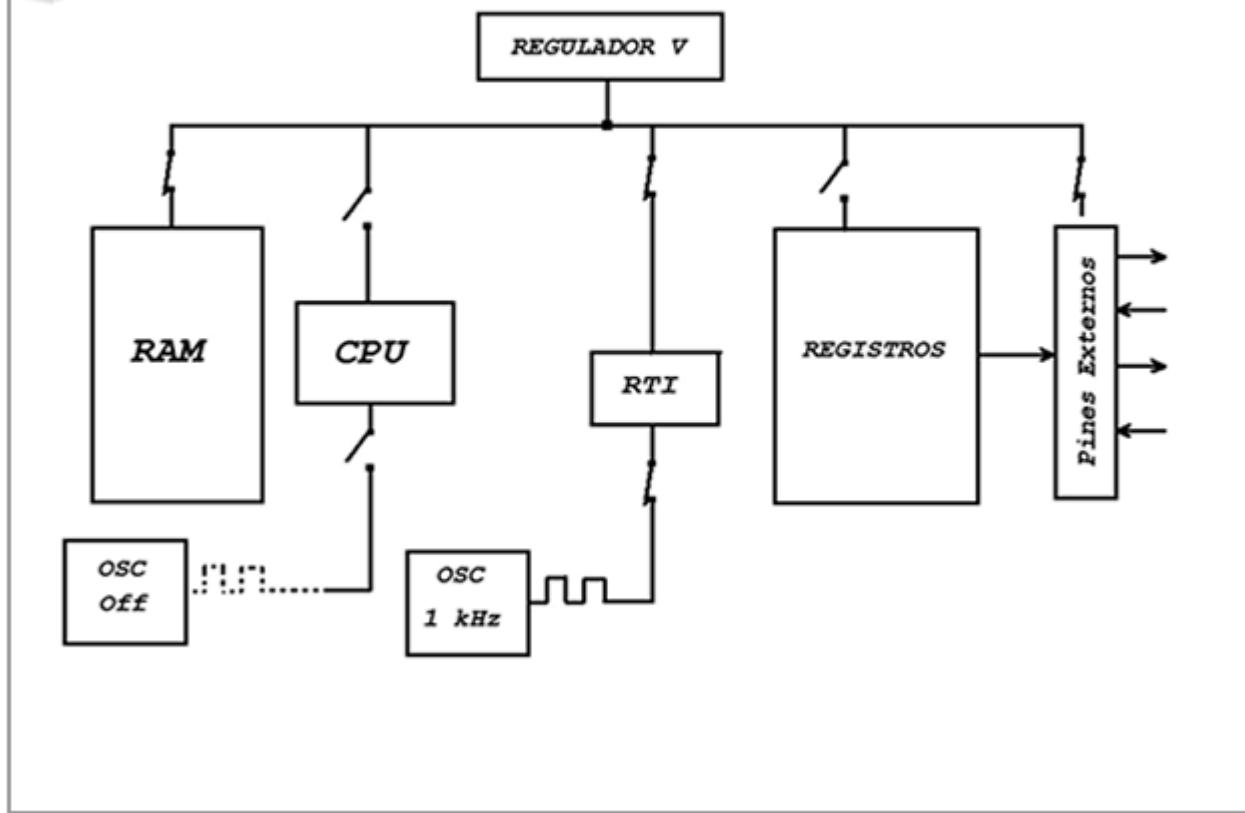


El programador deberá prestar atención a que los registros de entrada/salida durante el modo Stop2 queden efectivamente almacenados en la RAM, de lo contrario el sistema podría asumir el valor de Reset.





Modo Stop2 de bajo consumo



8.3.3 MODO "STOP 1"

Este modo provee el **más bajo consumo en standby**, causando que todos los circuitos internos del MCU sean apagados. Para seleccionar este modo, además de la ejecución de la instrucción **STOP**, deberá ponerse en 1 el bit **PDC** y en 0 el bit **PPDC** en el registro **SPMSC2** (ver Gráfico 8.5).

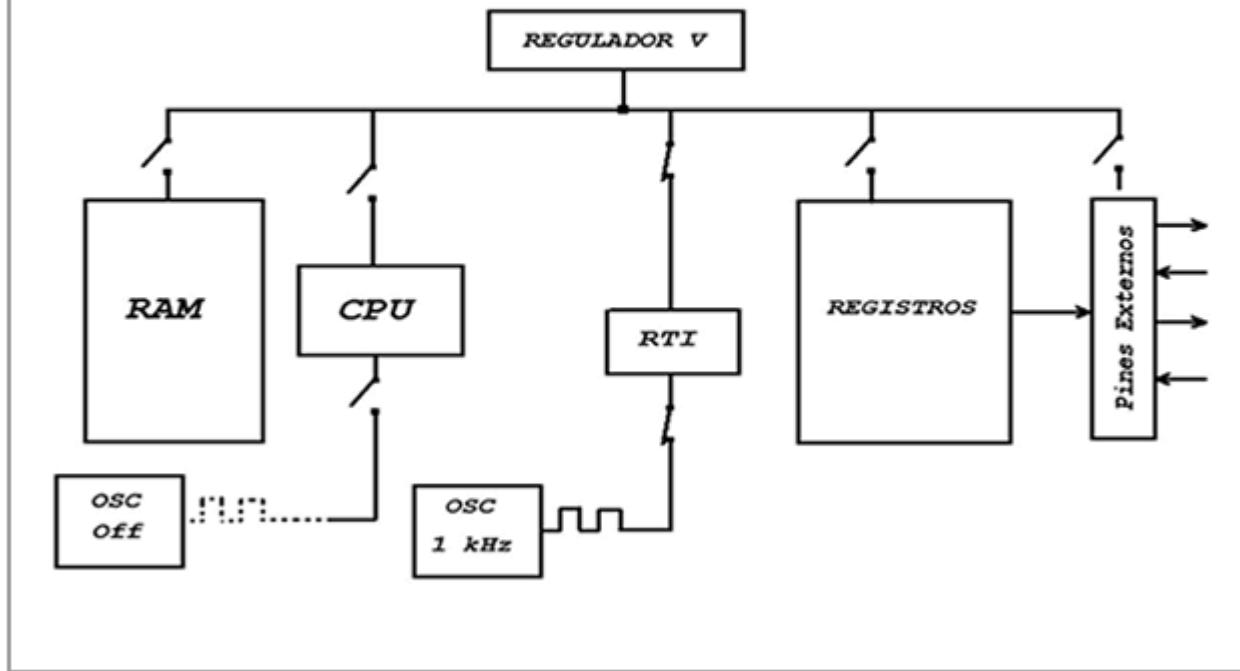
Una vez en modo **Stop1**, todos los circuitos que están siendo alimentados por el regulador de voltaje son apagados, y el regulador de voltaje es puesto en modo de *standby* de bajo consumo.

La salida de este modo se realiza ya sea por **Reset** o por **IRQ**, esta última señal deberá ser una **señal en bajo**, independiente de cómo fue configurada antes de entrar al modo.

Una vez salido del modo Stop1, el microcontrolador iniciará su secuencia como se hace por **POR (Power On Reset)**, la **CPU** tomará el vector de *Reset* e iniciará la ejecución de la aplicación.

**GRÁFICO
8.5**

Modo Stop1 de bajo consumo.



**GRÁFICO
8.6**

Promedio de consumo de energía en modo STOP

Energía

Promedio

Modo Run

Modo Run

Modo Run

Modo Stop

Modo Stop

Tiempo

8.4 OTROS MODOS DE BAJO CONSUMO

Los modos de bajo consumo tradicionales suspenden la ejecución de instrucciones a la **CPU** porque deshabilitan el oscilador, y solo puede ser recuperada su ejecución por el suceso de unas pocas interrupciones provenientes de algunos periféricos o un **reset**.

Sin embargo, en algunos estados de la aplicación, es deseable que la **CPU** ejecute algunas instrucciones de forma lenta, estados en los cuales no se requiere el 100% del desempeño por no estar realizando una labor muy exigente en tiempo o por requerir el monitoreo de eventos muy lentos.

En la aplicación de un sistema embebido como el que ejecuta un celular, el procesador central tiene asignadas tareas diversas que le exigen de forma diferente su desempeño, dependiendo del estado en el que se encuentre.

Estados y modos de bajo consumo de un sistema embebido.

GRÁFICO
8.7



Una forma de disminuir el consumo de energía consiste en que el procesador realice algunas de sus tareas con lentitud.

Los procesadores Flexis™ de Freescale™ cuentan con los modos de bajo consumo **Low Power Run** y **Low Power Wait**, los cuales hacen uso de la ejecución lenta y además mantienen muchos periféricos apagados por cortos períodos de tiempo.

Un celular en modo de reposo está en un modo de bajo consumo, con el display, la luz del teclado, el timbre apagados, entre otros periféricos; sin embargo, el procesador interno debe continuar su ejecución monitoreando el teclado, actualizando la fecha y la hora cada segundo, verificar si existe una llamada entrante, pero que le exige poco desempeño por tratarse de señales muy lentas.

En este caso el procesador no puede parar su ejecución por grandes lapsos de tiempo, pero puede ejecutar sus instrucciones de forma muy lenta, que permita que las tareas de monitoreo y actualización de variables internas se lleven a cabo.

En otros momentos en los cuales el usuario toma el celular y quiere realizar una marcación, el procesador tiene más exigencia en desempeño porque debe procesar instrucciones de forma más rápida, el modulo de display le exige mayor velocidad para procesar textos y en muchos casos imágenes, si se realiza acceso a algún navegador (WAP), deberá procesar las instrucciones que llegan para interpretarlas al usuario, si el usuario realiza acceso a algún juego, deberá no solo procesar el display, sino el audio, y así se van sumando aplicaciones que llevan al dispositivo a estados diferentes de exigencia en desempeño.

Es por este tipo de estados de diferente exigencia que en los recientes procesadores Flexis™ de Freescale™ se adicionan dos nuevos modos de bajo consumo denominados **Low Power Run** y **Low Power Wait**, que permiten tener un modo de bajo consumo con muchos de los periféricos apagados, pero continuando una ejecución de instrucciones muy lenta, o bien llevarlo por pequeños lapsos a un estado en el que no se ejecuten instrucciones reduciendo aun más el consumo, pero con varios periféricos aún encendidos.

8.4.1 Modo “Low Power Run” (**LPrun**)

Este modo de bajo consumo se recomienda cuando la aplicación requiere realizar un procesamiento muy sencillo que no requiera la velocidad completa del oscilador, en algunos casos que solo se requiere estar en un lazo que solo verifica algunos registros o eventos procedentes de algunos o todos los periféricos del **MCU**.

La gran ventaja del modo **LPrun** consiste en que puede ahorrar energía, aún cuando la **CPU** está en funcionamiento con todos los periféricos en operación.

En el modo **LPrun**, el regulador de voltaje interno es puesto en estado de *standby* (sin regulación). En este estado el consumo de potencia es reducido al mínimo permitido por la funcionalidad de la **CPU**, esto posibilita que el procesador continúe ejecutando instrucciones y realizando alguna labor en lugar de parar su ejecución.

La reducción se logra deshabilitando los osciladores que no son necesarios en los periféricos por medio de los bits en los registros **SCG1** y **SCG2**, y de esta forma el procesador queda en ejecución pero solo alimentando los periféricos que requiere.

Antes de entrar a este modo se deben ejecutar las siguientes condiciones:

FBELP (*FLL Bypassed External Low Power*) debe ser el modo de reloj seleccionado para el ICS (*Internal Clock Source*).

ICSC2[HGO] debe ser borrado.

La frecuencia del bus deberá ser menor a 125kHz.

El sensor de bajo voltaje (Low-Voltage Detect) deberá ser deshabilitado.

Para entrar al modo de **LPrun** solo se pone en 1 el bit LPR en el registro **SPMSC2** y para retornar al modo de ejecución normal se borra el mismo bit.

El bit **LPRS** en el registro **SPMSC2** es un bit de solo lectura usado para determinar si el regulador de voltaje esta en completa regulación o no. Una vez que el bit LPRS está en cero, indica que el regulador esta en completa regulación y que el MCU puede ahora ejecutar a la máxima velocidad.

Existe también la opción de retornar a completa regulación si alguna interrupción ocurre, lo cual se realiza poniendo en 1 el bit **LPWUI** en el registro **SPMSC2**. El **ICG** puede ser configurado para pasar a velocidad máxima una vez que se está en una **ISR** (*Interrupt Service Routine*).

En muchos casos se recomienda incluso apagar la memoria FLASH o de programa, previo paso a memoria **RAM** de un código corto

Tiene la limitación de no permitir en este modo la programación de memoria FLASH como memoria de datos.

Los límites de consumo de este modo están por los lados de **20uA**, cuando se ejecuta el código en memoria FLASH y de **9uA** cuando la ejecución del programa se realiza en la memoria RAM, con un oscilador de baja velocidad de 32kHz a un voltaje de alimentación de 3voltios.

8.4.2 Modo “Low Power Wait” (*LPwait*)



En equipos que realizan acciones muy sencillas, los modos de bajo consumo pueden incluir apagar la memoria flash, cuyo contenido previamente ha de vaciarse a la RAM.



El modo LPwait produce grandes

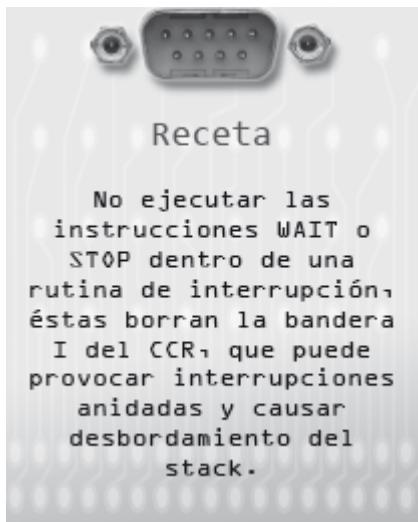
Al modo de *LPwait* se entra ejecutando la instrucción **stop** mientras que el microcontrolador está en modo de *LPwait*, el regulador de voltaje permanece *standby* igual que el modo *LPrun*.

En este estado el consumo de potencia es reducido al mínimo permitido para que la gran mayoría de los módulos mantengan su

de energía, pero en cambio mantiene el equipo en alerta para atender interrupciones con un período de respuesta muy corto.

funcionalidad. El consumo es reducido deshabilitando el oscilador a los módulos que no se usan borrando el bit correspondiente en el registro **SCGC**.

Las restricciones que aplican al modo *LPrun* aplican al modo **LPwait**. Es importante anotar que si el bit **LPWUI** se pone en 1 cuando la instrucción **STOP** es ejecutada, el regulador de voltaje retornará a su regulación completa cuando se salga del modo wait. El ICS puede ser configurado para que cambie la funcionalidad completa una vez que una rutina de ISR sea ejecutada, recuperando la **CPU** el máximo desempeño.



Si el bit de **LPWUI** es borrado cuando la instrucción **STOP** es ejecutada, el **MCU** retornará al modo de *LPrun*, continuando la ejecución de la **CPU** pero a velocidad lenta, conservando el ahorro de energía.

Cualquier *Reset* saca al procesador del estado de wait mode y borra el bit the **LPR** y retorna el **MCU** al modo normal de ejecución **RUN**.

El modo *LPwait* consume mayor energía que los modos de **Stop**, sin embargo, no tiene tiempo de recuperación como sí sucede en ese modo **stop**, una vez se reconoce una interrupción, el proceso de **stacking** se realiza de forma inmediata permitiendo que la interrupción sea atendida de forma muy rápida.

El límite de potencia para los procesadores de última generación Flexis está alrededor de los **5uA**.

8.5 CONSIDERACIONES EN EL DISEÑO DE UN SISTEMA EMBEBIDO

Los siguientes aspectos deberán ser considerados cuando el consumo de energía es importante en un sistema embebido:

Frecuencia de operación de la CPU

Recuerde que entre más alta sea la frecuencia del oscilador del microcontrolador, mas alto será su consumo de corriente. A la hora de un diseño se debe considerar la **frecuencia mínima** a la que el proyecto puede realizar su tarea más crítica.

Voltaje de operación

Si el procesador tiene un rango de voltaje amplio y se necesita ahorrar energía, es deseable verificar el **voltaje mínimo** al cual puede operar para disminuir el impacto en la ecuación de potencia. Sin embargo, no es recomendable estar muy cerca del límite mínimo, debido a que cualquier disminución en la fuente de alimentación, puede generar **reset** en el microcontrolador, provocado por el **LVI** (Sensor de bajo voltaje).



Una efectiva forma de ahorro energético consiste en trabajar el procesador a su voltaje y frecuencias mínimos el mayor tiempo posible, y sólo aumentarlo en procesos críticos.

Encender los dispositivos externos con baja corriente

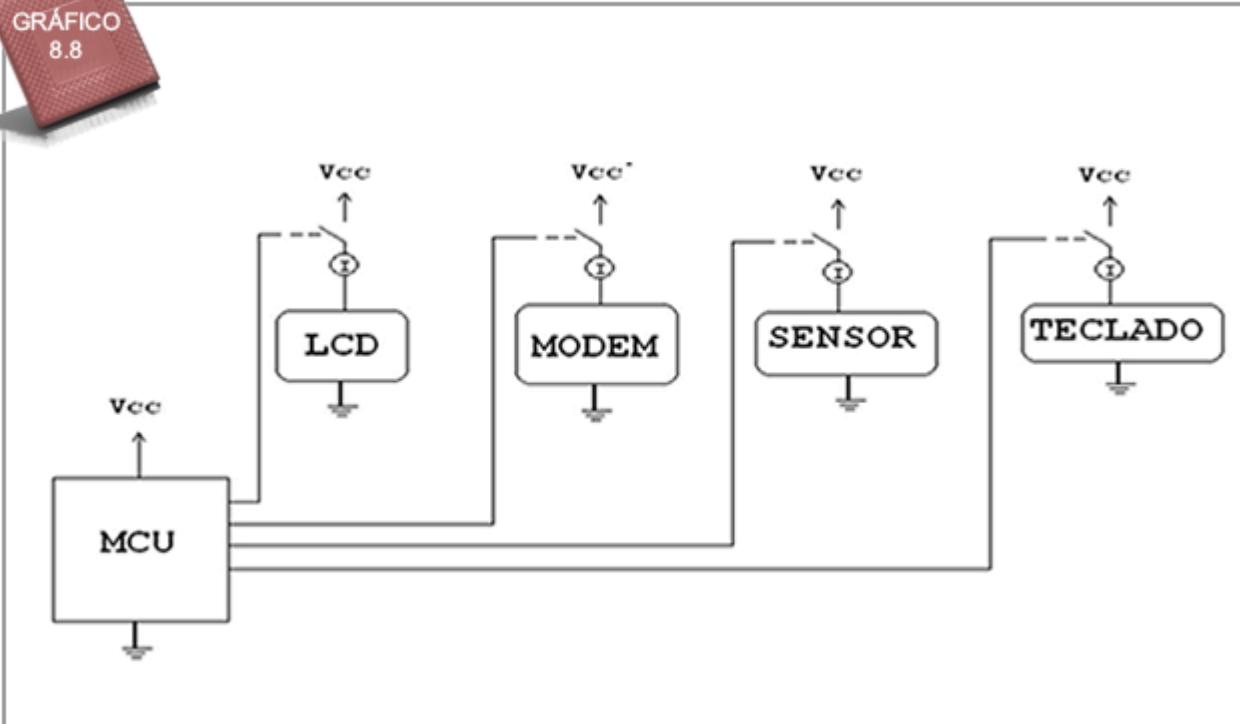
Al conectar LED, Relays, teclados, LCD, se debe intentar alimentarlos con la menor corriente posible, pero suficiente para encenderlos. En este caso es también importante seleccionar los dispositivos externos para que realicen su acción a menores voltajes y operen con menor corriente; se habla acá de las bobinas de los relevadores, los LED de bajo voltaje de operación y alta eficiencia, circuitos de interfaz de bajo voltaje.

En algunos casos también es recomendable controlar el voltaje de alimentación de los dispositivos externos, permitiendo apagar los dispositivos que en determinado estado del sistema no sean requeridos o que por condiciones de la fuente principal deban ser operados a menor corriente.

Esta técnica requerirá más pines de control del procesador debido a que el control de cada dispositivo requerirá como mínimo una señal adicional para controlar su apagado/encendido.



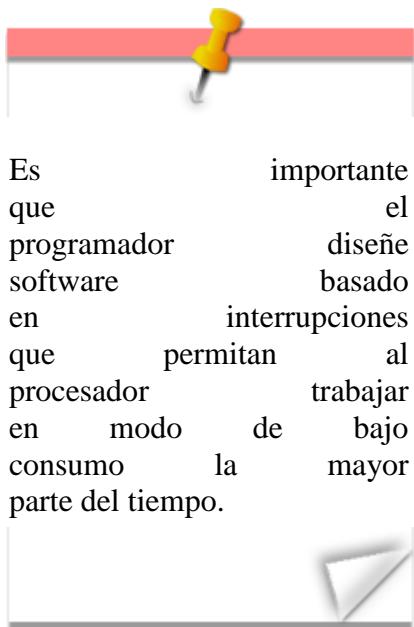
Control de apagado de dispositivos externos.



Pines NO usados en la aplicación

Los pines que no se usen en la aplicación no dejarlos sin conexión externa, recuerde que los pines bidireccionales de un microcontrolador por defecto están inicializados como entradas, en este caso deberán estar conectados a un nivel definido, sea cero (0) o uno (1), de forma directa o a través de una resistencia.

En caso que por espacio, o por facilidad del PCB, el pin no pueda ser conectado externamente, se recomienda inicializar dicho pin como salida, de esta forma su valor estará definido por el valor de salida del puerto.



Es importante que el programador software diseñe basado en interrupciones que permitan al procesador trabajar en modo de bajo consumo la mayor parte del tiempo.

Llevar el procesador a su menor estado de consumo

Cuando sea posible, llevar el microcontrolador a algunos de los modos de bajo consumo posible, **WAIT** o **STOP**, de esta forma la CPU detiene su ejecución y por ende el consumo de corriente de ejecución.

Deshabilitar el COP (*Watchdog*):

Si se desea un bajo consumo se recomienda deshabilitar el **COP** o **Watchdog** del procesador, a fin de evitar que éste suspenda el estado de bajo consumo del microcontrolador, dado que obliga un **reset**. Recuerde que el **COP** no puede ser habilitado y deshabilitado de forma dinámica, con lo que se deberá tomar la decisión desde un principio y para todo

el desarrollo del proyecto, si el uso del **Watchdog** se considera prioritario, se recomienda poner el procesador en bajo consumo y ayudarse de una interrupción periódica que invoque la rutina de llamado de borrado del contador del **COP**.

Deshabilitar periféricos NO usados

En caso de no requerir el sensor de bajo voltaje **LVI**, lo mejor será deshabilitarlo una vez que el procesador entre a algunos de los modos de bajo consumo.

Realizar software basado en interrupciones



Materiales adicionales en la web
Código fuente: el reloj de bajo consumo, para Freescale™

Un software basado en interrupciones permite que el procesador esté en modo de bajo consumo hasta que ocurra un evento, sea que se presione una tecla, que llegue un carácter por el puerto serial, que el procesador termine una conversión del ADC, o que se cumpla un tiempo.

Además de ahorrar energía permite que el procesador esté libre para otras tareas y no dedicado a una en particular.

EJEMPLO No. 31

El reloj de bajo consumo

Objetivo:

Escribir un código en C que permanece en estado de bajo consumo STOP el mayor tiempo posible, contabilizando tiempo real: año, mes, día, hora, minutos, segundos.

Cada minuto envía por el puerto serial la hora actualizada.

Usar las rutinas del modulo LCD.C para imprimir la hora actualizada cada minuto nuevo.

Solución:

```
***** Ejemplo 31 *****
// Reloj de Bajo Consumo
// Fecha: Abril 12,2009
// Asunto: RTC con impresión en display LCD
// y envío Serial
// Hardware: Sistema de desarrollo AP-Link
// FreescaleTM.
// Version: 1.0 Por: Gustavo A. Galeano A.
//*****
#include <hidef.h>
#include "derivative.h"
#include "serial.h"
#include "lcd.h"
#include <stdlib.h>
#include <time.h> //Funciones de tiempo
#define INPUT_1_Press() !PTD_PTD1
#define INTS_TO_1SEG 37 //Nro de Ints TBM para 1 seg
// variables globales
struct tm *horaActualPtr; /*apuntador a la estructura de la hora*/
struct tm horaActualStr; /*estructura con hora actual en campos*/
char buffer[BUF_SIZE]; /*buffer para recepción de datos seriales*/
time_t segAct,segAnt; /*seg actuales desde 1 Enero,1970*/
const unsigned char moninit[12] =
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
static char buf[19]; /*buffer para formato de la fecha*/
time_t horaActual;
char *timeMsgPtr; /*apuntador a estructura de hora en ascii*/
void Led_Neg(void){ /*Funciones encendido y apagado de LEDs*/
    DDRC_DDRC3 = 1;
    PTC_PTC3 ^= 1; /*Invierte estado Led ON/OFF*/
}
void Led_Out2_ON(void){
    DDRC_DDRC2 = 1;
    PTC_PTC2 = 1;
}
void Led_Out2_OFF(void){
    DDRC_DDRC2 = 1;
    PTC_PTC2 = 0;
```

```

}

/*Función de recepción de dato via serial*/
void Serial_RxOperando(int *operandoPtr){
    Serial_RecibirCmd(buffer);
    *operandoPtr = atoi(buffer);
}

/*Función de Inicialización del registros del MCU*/
void Mcu_Init(void){
    CONFIG1 =
        (CONFIG1_LVIRSTD_MASK|CONFIG1_LVIPWRD_MASK|CONFIG1_LVIPWRD_MASK|
        CONFIG1_COPD_MASK|CONFIG1_STOP_MASK);
    /*deshabilita el COP y habilita la instrucción STOP*/
    CONFIG2 =
        (CONFIG2_STOP_ICLKDIS_MASK|CONFIG2_OSCCLK1_MASK|
        CONFIG2_STOP_XCLKEN_MASK|CONFIG2_SCIBDSRC_MASK);
    /*habilita oscilador en modo STOP y SCIBDSRC*/
}

void RTC_Init(void){/*Inicialización del RTC*/
    TBCR_TBR0 = 0;
    TBCR_TBR1 = 0;
    TBCR_TBR2 = 0; /*define prescaler del TBR /1*/
    TBCR_TBON = 1; /*Enciende Timer*/
    TBCR_TACK = 1; /*borra bandera TBIF*/
    TBCR_TBIE = 1; /*habilita interrupción de TBM*/
}

/*ISR de TimeBase, ocurre cada 26.67mSeg*/
interrupt 22 void TimeBase_ISR(void){
    static unsigned int cntInts;
    cntInts++;
    if(cntInts >=INTS_TO_1SEG){
        cntInts = 0;
        segAct++;
        Led_Neg();
    }
    TBCR_TACK = 1; //borra bandera TBIF
}

/*Función para creación de formato en buffer*/
void put2d(char * cp,uchar i,char cl, char ct){
    *cp = ct;
    *--cp = i%10 + '0';
    if(i /= 10){ *--cp = i + '0';}
    else{ *--cp = cl;}
}

static dylen(unsigned yr){/*Retorna dias del año*/
    if(yr%4) return(365);
    return(366);
}

/*Función de retorno de apuntador a estructura

```

```

con fecha a partir de los segundos actuales*/
struct tm * localtime(const time_t * tp){
unsigned int day,mon,yr,cumday;
time_t t;
static struct tm tim;
unsigned char monlen[12];
for(mon = 0 ; mon != 12 ; mon++) monlen[mon] = moninit[mon];
t = *tp;
tim.tm_sec = t % 60L;
t /= 60L;
tim.tm_min = t % 60L;
t /= 60L;
tim.tm_hour = t % 24L;
day = (unsigned int)(t / 24L);
tim.tm_wday = (day + 4) % 7;
yr = 70;
cumday = 0;
while((cumday += dylen(yr)) <= day) yr++;
tim.tm_year = yr;
cumday -= dylen(yr);
day -= cumday;
tim.tm_yday = day;
cumday = 0;
mon = 0;
if((yr%4) == 0){monlen[1] = 29;}
else{monlen[1] = 28;}
while((cumday += monlen[mon]) <= day) mon++;
cumday -= monlen[mon];
day -= cumday;
tim.tm_mday = day + 1;
tim.tm_mon = mon;
return &tim;
}
/*Función de retorno de apuntador a cadena en
formato DiaSem, Mes DiaMes hora:min*/
char * asctime(const struct tm * tim){
char * s, * cp;
unsigned char i;
s = &"DomLunMarMieJueVieSab"[tim->tm_wday*3];
i = 3;
cp = buf;
do
    *cp++ = *s++;
while(--i);
    *cp++ = ',';
s = &"EneFebMarAbrMayJunJulAgoSepOctNovDic"[tim->tm_mon*3];
i = 3;
do

```

```

*cp++ = *s++;
while(--i);
buf[7] = ' ';
put2d(buf+10, (unsigned char)tim->tm_mday, ' ', ' ');
put2d(buf+13, (unsigned char)tim->tm_hour, '0', ':');
put2d(buf+16, (unsigned char)tim->tm_min, '0', ' ');
return (buf);
}
/*Retorna los segundos actuales de la struct tm*/
time_t mktime(struct tm *tiempo){
static time_t tiempoL;
unsigned char ano, mes;
unsigned int dias;
ano = (unsigned char)tiempo->tm_year;
mes = (unsigned char)tiempo->tm_mon;
dias = 0;
while(ano > 70){
ano--;
dias = dias + dylen(ano);
} /*dias tiene acumulado el número de días de años anteriores */
while(mes > 0){
mes--;
dias += moninit[mes];
} /*se le adicionan los días de los meses vencidos*/
if ((tiempo->tm_mon > 2) && ( (tiempo->tm_year % 4) == 0 )){ dias++;}
/*si el mes es mayor a febrero y es bisiesto se suma un dia mas*/
dias = dias + tiempo->tm_mday-1;
tiempoL = dias * 24L; /* convertido a horas */
tiempoL = (tiempoL + (unsigned)tiempo->tm_hour) * 60L; /* convertido a
minutos */
tiempoL = (tiempoL+(unsigned)tiempo->tm_min) * 60L; /* convertido a segundos
*/
tiempoL = (tiempoL + (unsigned)tiempo->tm_sec);
return tiempoL;
}
void SetRTC(void){ /*Inicialización de Fecha y Hora vía Serial*/
if(!INPUT_1_Press()){
do{
Serial_SendMsg("\r\nAno(1970..2100)>");
Serial_RxOperando(&horaActualStr.tm_year);
horaActualStr.tm_year -=1900;
}while(horaActualStr.tm_year < 70);
do{
Serial_SendMsg("\r\nMes(1...12)>");
Serial_RxOperando(&horaActualStr.tm_mon);
horaActualStr.tm_mon--;
}while((horaActualStr.tm_mon<0) || (horaActualStr.tm_mon >= 12));
do{

```

```

Serial_SendMsg("\r\nDia(1...31)>");
Serial_RxOperando(&horaActualStr.tm_mday);
}while(horaActualStr.tm_mday > 31);
do{
    Serial_SendMsg("\r\nHora(0...24)>");
    Serial_RxOperando(&horaActualStr.tm_hour);
}while(horaActualStr.tm_hour > 23);
do{
    Serial_SendMsg("\r\nMin(0...59)>");
    Serial_RxOperando(&horaActualStr.tm_min);
}while(horaActualStr.tm_min > 59);
do{
    Serial_SendMsg("\r\nSeg(0...59)>");
    Serial_RxOperando(&horaActualStr.tm_sec);
}while(horaActualStr.tm_sec > 59);
}while{ // si la tecla no esta presionada una fecha cualquiera
    horaActualStr.tm_year = 108;//2008
    horaActualStr.tm_mon = 11; //Dic
    horaActualStr.tm_mday = 1; //dia 1
    horaActualStr.tm_hour = 20; //hora 8 pm
    horaActualStr.tm_min = 30; //min 30
    horaActualStr.tm_sec = 0; //0 segundos
}
}
/*Verificación de cambio de fecha*/
char TimeChange(void){
static int minAnt;
if(segAnt == segAct) return 0;
else{
    segAnt = segAct;
    horaActualPtr = localtime (&segAct);
    horaActualStr.tm_sec = horaActualPtr->tm_sec;
    horaActualStr.tm_min = horaActualPtr->tm_min;
    horaActualStr.tm_hour = horaActualPtr->tm_hour;
    horaActualStr.tm_mday = horaActualPtr->tm_mday;
    horaActualStr.tm_mon = horaActualPtr->tm_mon;
    horaActualStr.tm_year = horaActualPtr->tm_year;
    horaActualStr.tm_wday = horaActualPtr->tm_wday;
    if(INPUT_1_Press() || (minAnt != horaActualPtr->tm_min)){
        minAnt = horaActualPtr->tm_min;
        return 1;
    }else{
        return 0;
    }
}
void main(void){ /*Función principal*/
    Mcu_Init();
}

```

```

Serial_Init(); //initializa módulo SERIAL
Lcd_Init(); //initializa módulo LCD
RTC_Init(); //initializa módulo TBM
EnableInterrupts; /* enable interrupts */
Serial_SendMsg("\r\n***** ALFAOMEGA GRUPO EDITOR *****\r\n");
Serial_SendMsg(" www.alfaomega.com.mx \r\n");
Serial_SendMsg("---- Ejemplo RTC de Bajo Consumo ----\r\n");
PutsD("Reloj Bajo Consumo");
SetRTC(); //cuadrar fecha y hora
segAct = mktime(&horaActualStr); //initializa la variable segAct
for(;;) {
    if(TimeChange()){
        timeMsgPtr = asctime(horaActualPtr);
        Serial_SendMsg("\r\n");
        Serial_SendMsg(timeMsgPtr);
        Lcd_Goto(22);
        PutsD(timeMsgPtr);
    }
}
}

```

Materiales adicionales en la



Información sobre los modos de bajo consumo de los procesadores HC(S)08 y Flexis™.

Discusión:

Se debe inicializar el registro CONFIG1 en un solo ciclo, debido a que como lo aclara la hoja de datos del AP16A este registro solo puede ser escrito una sola vez después de RESET.

También en el registro CONFIG2 es necesario encender el bit CONFIG2_STOP_XCLKEN, para que una vez el microcontrolador en modo STOP, el oscilador externo permanezca encendido aún en este modo y no se suspenda la contabilización del tiempo.

La toma inicial de datos para cuadrar la hora y para visualizar la hora por el puerto serial se verá así:

```
---- ALFAOMEGA GRUPO EDITOR -----  
www.alfaomega.com.mx  
---- Ejemplo RTC de Bajo Consumo ----  
Año(1970...2100)>2008  
Mes(1...12)>11  
Dia(1...31)>13  
Hora(0...24)>21  
Min(0...59)>50  
Seg(0...59)>00  
Jue, Nov 13 21:50
```

RESUMEN DEL CAPÍTULO

El bajo consumo de energía es uno de los aspectos cada vez mas importantes a considerar en el desarrollo de las aplicaciones embebidas actuales, permiten que los equipos portables sean más pequeños, más compactos y requieran menor mantenimiento e intervención del usuario.

Los estados en los que se encuentre una aplicación embebida determinan el modo en el que el procesador pueda estar, dependiendo además de los periféricos que se requiera que estén activos en ese estado. Los modos generales de bajo consumo **WAIT** y **STOP**, permiten administrar el consumo dependiendo si los periféricos internos requieren o no estar activos; sin embargo, se ha determinado que muchas aplicaciones requieren estados intermedios de funcionamiento y que no paren la ejecución de la máquina, en algunos casos no se requiere el 100% del desempeño de la máquina y el regulador de voltaje 100% activo.

Se ahorra energía con la administración independiente de los periféricos internos y externos al procesador central de una aplicación embebida, y es donde el software juega un papel importante en la administración de estos periféricos.

Cuando se realiza un diseño embebido con limitaciones de consumo deberá hacerse de tal forma que opere al menor nivel de voltaje de alimentación, por el lado del software deberá calcularse la frecuencia de operación mínima de la CPU para que la operación más crítica o señal más rápida posible en el sistema pueda ser ejecutada, de esta forma se garantiza que el consumo será menor.

Al mismo tiempo, establecer los estados posibles de la maquina y la posibilidad de apagar o encender diferentes periféricos en los distintos estados que tenga el

sistema.

Por último, establecer un funcionamiento basado en interrupciones y no en ejecuciones continuas (*polling*), que permita que el procesador pueda estar en modos **WAIT** o **STOP** por instantes de tiempo más largos.

PREGUNTAS Y EJERCICIOS PROPUESTOS

Optimizar el consumo del ejemplo “Reloj de bajo consumo”, usando las interrupciones de la interfaz serial SCI para envío y recepción de datos, usando el modo **WAIT**.

¿Qué efecto tienen en el consumo las siguientes variables de un sistema embebido: voltaje de alimentación, frecuencia de oscilador, capacitancia de carga de los pines de entrada/salida, temperatura en el ambiente de operación del procesador, uso del modulo COP y uso del modulo LVI?

¿Qué utilidad tiene el modo de bajo consumo **LPwait** frente al modo **WAIT**, considerando que ambos no realizan ejecución de instrucciones de la CPU?

¿En cuáles casos se puede considerar útil el uso del modo de bajo consumo **LPrun**?

¿Qué desventajas tiene el control de periféricos externos para el control de consumo de energía? Mencione al menos tres (3).

Realizar un análisis de los estados y los modos de bajo consumo posibles de una cámara digital de fotografía.

INTRODUCCIÓN

Uno de los problemas más comunes en el desarrollo de un proyecto que involucra microcontroladores, radica en el hecho de tener que conocer muy bien la máquina que se está usando, a fin de configurar de manera adecuada sus periféricos.

Y aunque las hojas de datos son muy exactas y claras en sus especificaciones, el componente humano que digita el código no está exento de cometer errores, los cuales demandan tiempo de corrección y exigen un gran nivel de concentración.

Así por ejemplo, si se requiere configurar un puerto serial, un temporizador o un ADC con determinado comportamiento, se tendrán que configurar varios registros del periférico, su modo de operación, sus interrupciones, realizar el borrado de las respectivas banderas, etc.

El capítulo presenta una alternativa que puede ser usada de forma gratuita por usuarios de tres tecnologías (hasta ahora) como son las de Freescale™, Fujitsu™ y National Semiconductor™ para la configuración rápida de los periféricos internos, por medio de un ambiente gráfico de fácil y rápido manejo, eliminando de esta forma la posibilidad de cometer errores al momento de transcribir el manual de especificaciones del microcontrolador al código en C.

Se presentan los componentes y ventanas típicas del ambiente y por medio de tres ejemplos se muestra de forma práctica el uso básico, la creación de componentes de software y el uso un poco más avanzado que incorpora varios elementos de hardware y de software.

Al final se listan algunas consideraciones sobre el uso del “Processor Expert™” las cuales deben estar en la mente del programador del sistema para que dichas herramientas le ayuden a generar un código útil y eficiente.

9.1 “PROCESSOR EXPERT™”



La configuración de los periféricos es parte fundamental en todo diseño para sistemas embebidos, para ello el programador deberá considerar cada uno de los módulos internos de la máquina.

La configuración de periféricos se

En general, es necesario considerar para cada uno de los módulos internos de la máquina muchos aspectos: si alguno de ellos no es configurado adecuadamente el sistema no funcionará de la forma esperada, y arreglar el impasse ocasionará inversiones adicionales de tiempo del programador, largas horas de lectura de las hojas de datos o varios ciclos de prueba y error que permitan encontrar el problema.

En el caso del módulo temporizador (**TIMER**), si se quiere una interrupcion periódica de 100mSeg se deberá considerar la frecuencia del oscilador, configurar registros de preescalado, configurar la función de inicialización (**Timer_Init**) y la función de interrupción (**Timer_Isr**), realizar el borrado de las banderas de interrupción, etc.

El “*Processor Expert™*”, es una herramienta de software creada por la compañía checa UNIS, incorporada en el paquete de Codewarrior® IDE. Permite configurar la capa de bajo nivel de cualquier microcontrolador de la marca Freescale™, en un ambiente familiar y unificado para usuarios de MS Windows usando herramientas **RAD** (*Rapid Application Design*).

Esta herramienta se puede habilitar de forma opcional en el paso número 8 de creación de un proyecto en lenguaje C (Capítulo 3) con el Codewarrior®, una vez seleccionado, brinda al programador la posibilidad de elegir los módulos internos que requiere habilitar con su respectiva funcionalidad.

Esta facilidad permite mediante un ambiente gráfico, configurar todos los módulos internos del microcontrolador y generar código en C abierto con sus comentarios respectivos, el cual puede ser modificado por el programador, usado en otro proyecto o compilado para ser ejecutado en el procesador.

El “*Processor Expert™*” tiene la ventaja de facilitar la configuración de un sistema sin necesidad de conocer en detalle la máquina, sus registros y cada uno de sus bits, además de facilitar el trabajo en equipo.



Un poco más avanzado es el tema de creación de componentes de software usando el “*Bean Wizard™*” el cual proporciona la flexibilidad de crecer las prestaciones del proyecto a futuro usando “*Processor Expert™*”.

Esto conlleva un menor tiempo de desarrollo, fácil migración entre procesadores y concentra al programador en las capas superiores de un proyecto.

Sin embargo, solo es recomendado para realizar pruebas específicas de la máquina o como ayuda para configurar algún periférico, debido a que genera código redundante que es crítico en aplicaciones con procesadores de capacidad limitada de memoria.

facilita con el uso de un software adecuado; entre ellos el “Processor Expert™” es uno de los más completos y sencillos de manejar.



9.2 EL CONCEPTO DEL “BEAN”



Un “**bean**” es un componente de software ya listo para usar, que encapsula la inicialización y funcionalidad de un elemento básico de un microcontrolador, tales como la CPU y sus periféricos (timer, SCI, SPI, I2C, ADC, etc.).



Un “**bean**” es un componente listo para usar que genera una secuencia de código con base en propiedades. Se diferencia de las librerías en que no son estándar y están orientadas a manejar hardware o rutinas especializadas de software embebido

Proveen una capa de abstracción de hardware HAL¹, que facilita la configuración y migración entre dispositivos a través de un interfaz gráfico al usuario GUI². Cada “**bean**” puede ser accesado por medio de *propiedades, métodos y eventos*.

Propiedades: este campo permite especificarle al “**bean**” definiciones precisas del periférico a usar, como son el nombre, las velocidades de la línea serial, los períodos de interrupción, etc. El “**Processor Expert™**” verifica en tiempo real que las configuraciones del usuario sean válidas y provee algunas recomendaciones básicas como valores permitidos máximos o mínimos.

Métodos: permite definir macros y funciones que modifican el comportamiento de la aplicación en tiempo de ejecución. Dependiendo del periférico y de su funcionalidad, se sugieren funciones que están habilitadas/deshabilitadas en el campo, y al configurarlas pueden cambiarse sus atributos para que aparezcan o no en el código generado; además, posibilitan la optimización del código, ya sea para aumentar la velocidad de ejecución o disminuir el tamaño del código generado.

Eventos: provee funciones de llamado cuando suceden cambios importantes en el “**bean**”, con las cuales el programador agrega código de la capa de aplicación.

Los **beans** pueden ser de tipo software exclusivamente para manejar periféricos del microcontrolador o hardware externo, en cualquiera de los casos, el acceso desde la capa de aplicación se realiza por medio de procedimientos creados y probados.

1 HAL (Hardware Abstraction Layer).

2 GUI (Graphical User Interface).

9.3 LIBRERÍA “BEANS”



Una librería de “beans” es un componente del “Processor Expert™” que contiene un set de “beans” de alto nivel, listos para ser usados por el programador. Muchas de las librerías vienen incluidas al instalar el paquete del Codewarrior®, otras pueden ser creadas por el mismo programador para permitir a otros su uso en futuros proyectos, o bien pueden ser bajadas de la página del “Processor Expert™”. www.processorexpert.com.



Entre las ventajas del “Processor Expert™” está el no generar tanto código redundante como lo hacen otros programas.



En la pagina www.processorexpert.com se pueden encontrar extensas librerías de beans, que el programador podrá descargar y usar en sus proyectos.

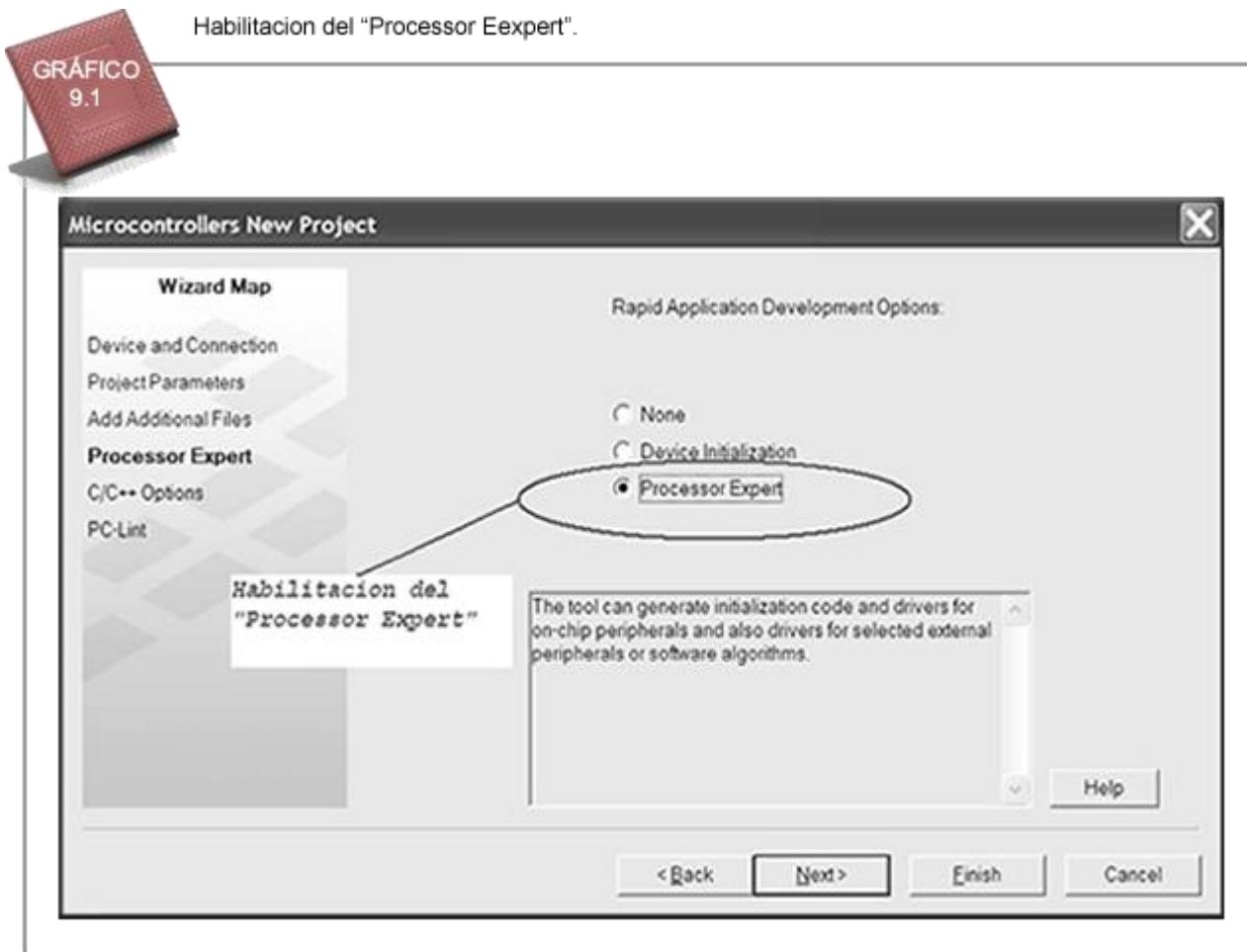


En la misma página se puede encontrar mayor información y detalles de la creación de proyectos usando esta excelente herramienta de evaluación y programación para los microcontroladores de Freescale™.

Además de los “beans” que ya trae el Codewarrior®, en la página se pueden bajar otros adicionales para manejo de periféricos externos como son los acelerómetros, conversores ADC y DAC vía SPI de las compañías Linear technology y Maxim; controladores digitales de audio, operaciones aritméticas complejas como son el CRC (Código de Redundancia Cíclica), encriptación de datos, FFT (*Fast Fourier Transform*) , Filtros FIR, manejo de display gráfico, teclado matricial, reproductores de archivos WAV, entre otros.

9.4 CREACIÓN DE PROYECTOS USANDO EL “PROCESSOR EXPERT™”

La creación de proyectos usando “Processor Expert™”, se realiza hasta el paso 7, de la misma forma que la creación de un proyecto en C usando Codewarrior (Capítulo 3), en el paso 8 se elige la opción de incluir el “Processor Expert™”. Habilitación del “Processor Expert™”.

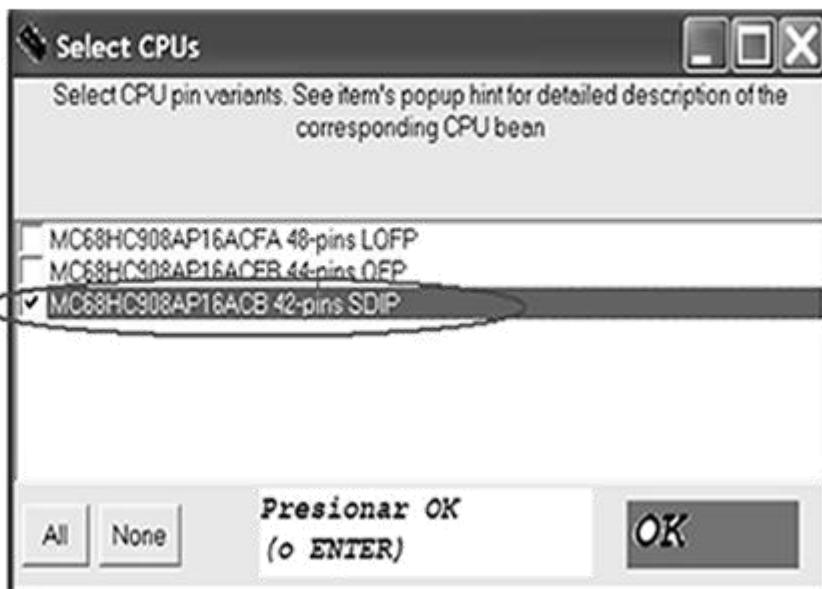


Una vez seleccionado el botón “Next”, se procede con los pasos 9 (Opciones de C/C++) y 10 (Uso del **Pc_Lint**) que continúan con el mismo significado para el proyecto. Al presionar “Finish” en la última ventana, la creación muestra una nueva ventana llamada “Select CPUs”, que permite seleccionar el empaque final del microcontrolador usado.



Ventana de selección del empaque del microcontrolador.

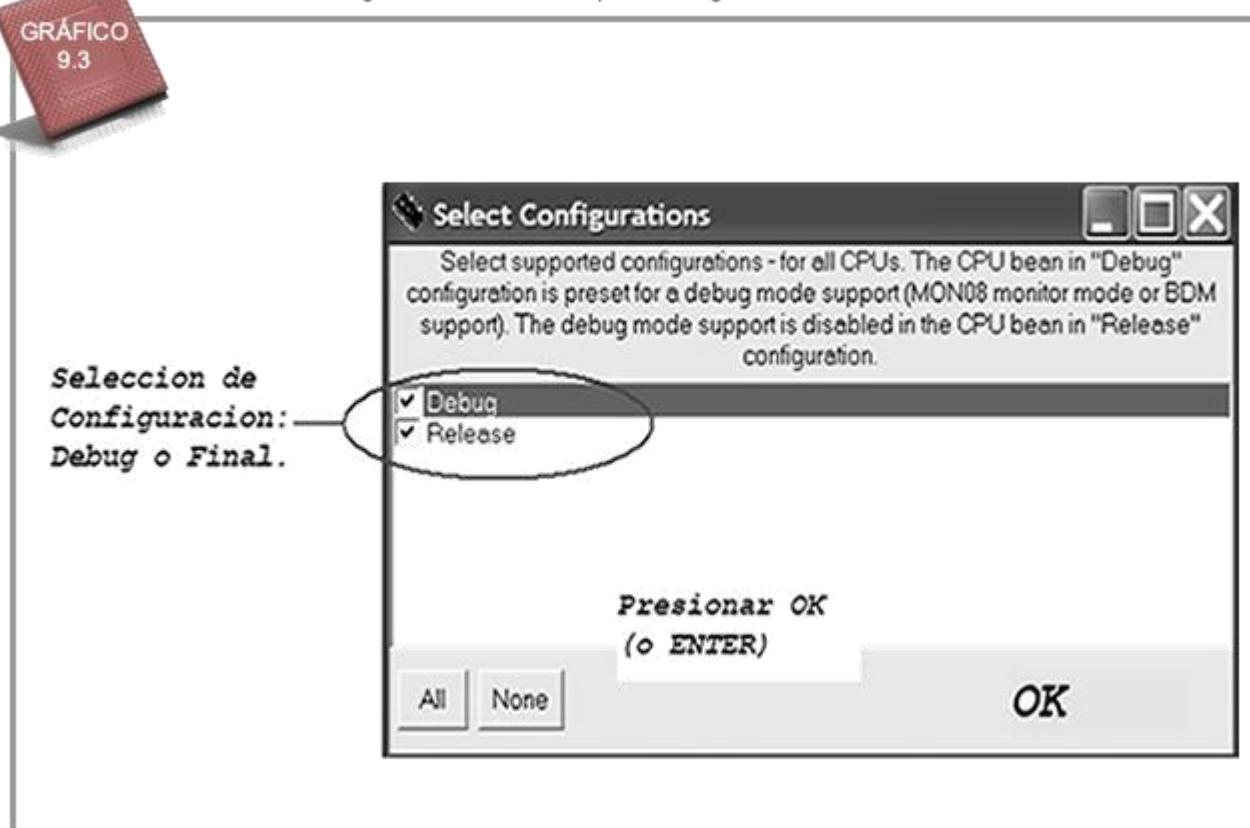
GRÁFICO
9.2



Por último, la ventana de “*Select Configurations*”, permite elegir el modo de depuración; si solo el modo “**Debug**” es seleccionado, el modo **MON8** o **BDM** es soportado (es lo recomendado), mientras que en el “**Release**” el modo de depuración es deshabilitado.



Ventana de configuración de versión para Debug o final.



Una vez presionado el botón de OK, el “Processor Expert™” genera código básico, (como el StartUp) y el proyecto ahora presenta tres ventanas básicas que permiten configurar y adicionar cada uno de los “beans” posibles.

9.5 Ventanas del proyecto con “Processor Expert™”

9.5.1 La ventana “bean inspector”



En los bean es posible configurar las opciones, métodos y eventos dependiendo del tipo de periférico a utilizar.



La ventana “**bean inspector**” permite configurar las diferentes opciones, métodos y eventos de cada “**bean**”, estos pueden variar dependiendo de cada periférico y de cada microcontrolador en particular. En la parte inferior de la ventana, se puede elegir entre tres tipos de nivel para el manejo de la ventana, dependiendo del conocimiento que el programador tenga de cada “**bean**” (*ver Gráfico 9.4*).

BASIC (básico): permite el manejo más sencillo del “**bean**”, presentando las opciones fundamentales del componente.

ADVANCED (avanzado): habilita opciones más completas del “**bean**”, que otorgan al programador mayor flexibilidad, pero a su vez un mejor nivel de entendimiento del componente.

EXPERT (experto): presenta de forma completa todas las opciones y detalles del “**bean**”, permite la máxima flexibilidad de configuración, pero también exige de un buen conocimiento del periférico.

Puede usarse el menú **Help** ◊ **Help On Bean**, para consultar la ayuda de cada uno de los componentes.

GRÁFICO
9.4

Ventana "bean inspector" del módulo SCI.

**Propiedades**
del "bean"**Métodos**
del "bean"**Eventos**
del "bean"**Nombre**
del "bean"

La primera configuración que el programador requiere realizar en cualquier proyecto por sencillo que sea, es el de la **CPU**. Este “bean” es incluido automáticamente al iniciar un proyecto con **“Processor Expert™”**. Su función consiste en realizar la definición del tipo de oscilador a utilizar, sea interno o externo, la frecuencia del oscilador, los modos de bajo consumo habilitados y las opciones del sistema completo de la **CPU** y su oscilador (*ver Gráfico 9.5*).

Nótese que para los ejemplos prácticos que se están ejecutando con el uso del sistema AP-Link, el oscilador seleccionado es externo y su frecuencia es de 9.8304MHz. A partir de este, se pueden habilitar o no opciones de la **CPU**, como son el manejo de la instrucción **STOP** con sus respectivas opciones de recuperación, y manejo de interrupciones provenientes de este periférico.



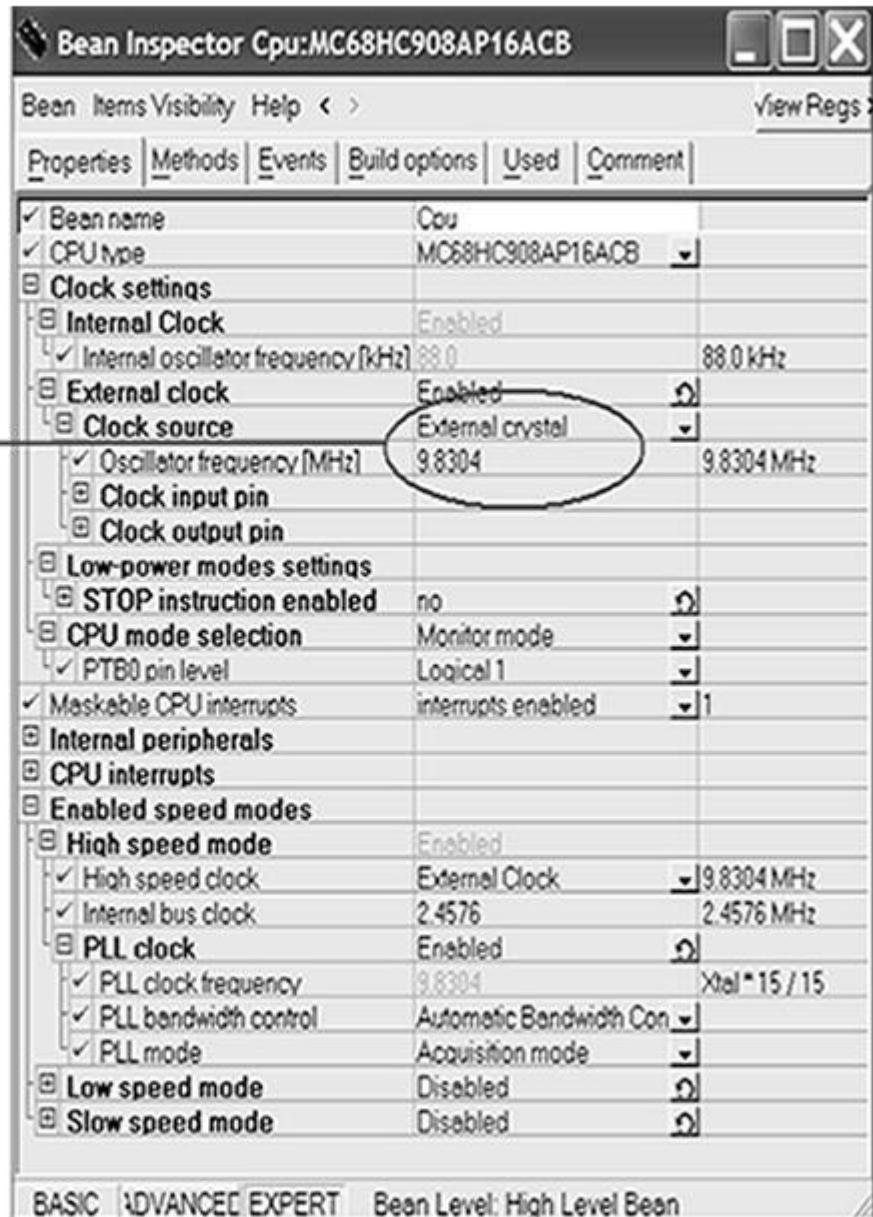
El bean llamado **CPU** se crea por defecto cuando el programador inicia un proyecto; entre sus funciones está definir la frecuencia del oscilador y habilitar los sistemas de bajo consumo.



GRÁFICO
9.5

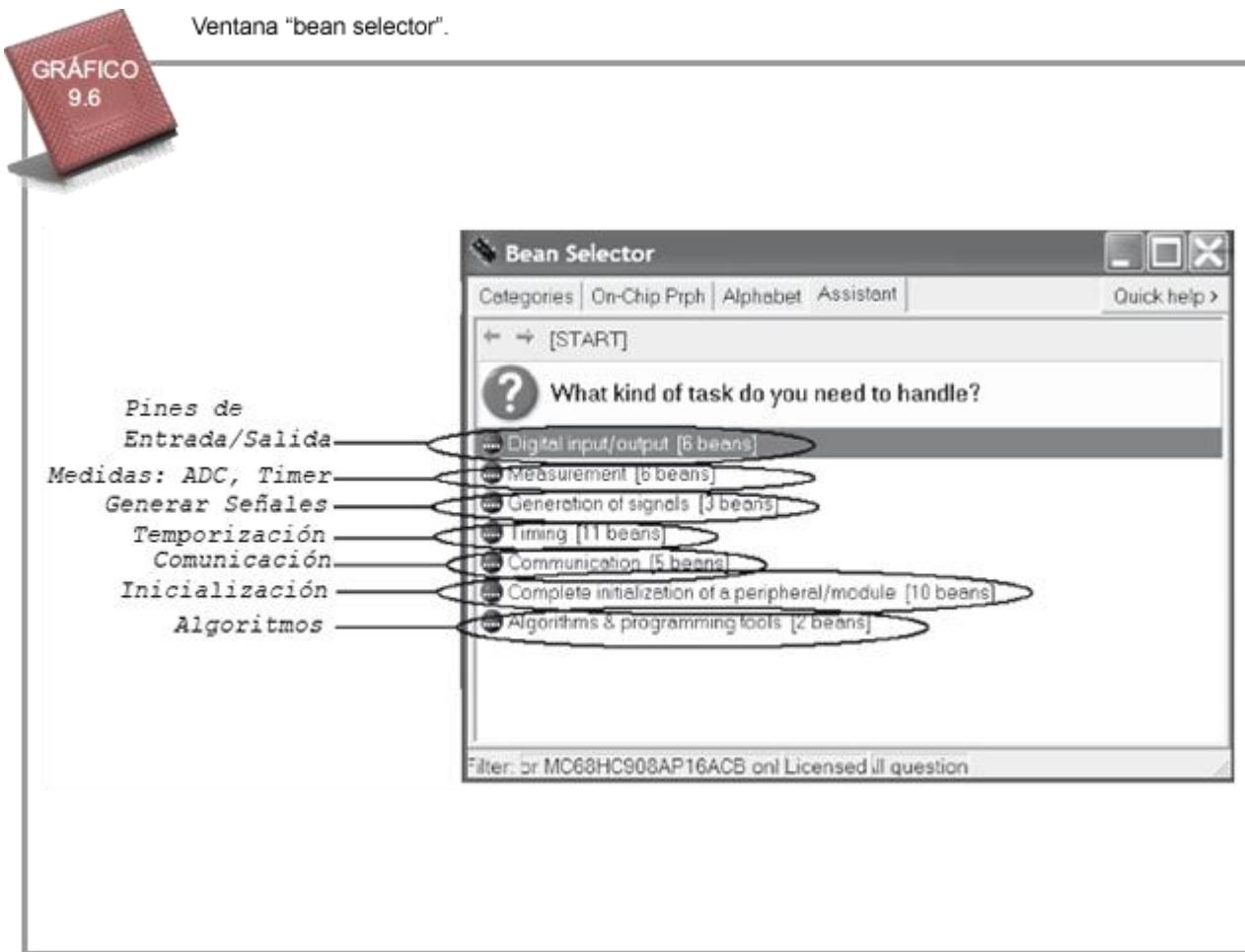
Ventana "bean inspector" de CPU.

Definicion del
Oscilador
Externo y
Frecuencia

9.5.2 La ventana "bean selector"

Presenta un listado de los “beans” de la parte específica seleccionada y las tareas propias que pueden ser incluidas en el proyecto (paso 4 de la creación de proyectos en C en Codewarrior®, Capítulo 3) (*ver Gráfico 9.6*).



Esta ventana permite, mediante sus diferentes opciones, organizar la lista de “beans” ya sea por:

- Categorías (**Categories** Tab).
- Periféricos internos (**On-Chip Prph** Tab).
- Alfabéticamente (**Alphabet** Tab).
- Asistente de selección (**Assistant** Tab).

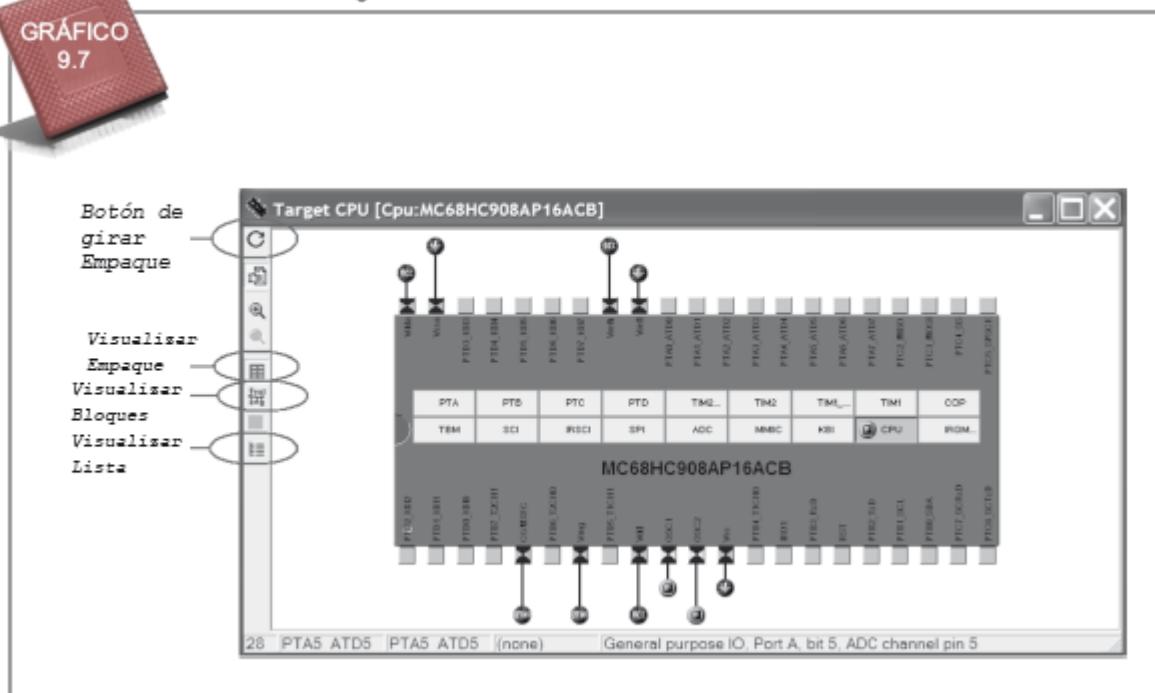
9.5.3 La ventana “Target CPU”

La ventana “**Target CPU**” muestra el componente seleccionado con sus periféricos y pines de forma más real, tal cual la distribución del microcontrolador físico.

Es posible tener diferentes opciones de visualización usando los botones en la parte izquierda de la ventana, ya sea visualizar el empaque (como aparece en el *Gráfico 9.7*), o visualizar los periféricos en diagrama de bloques, o bien, en lista alfabética.



Ventana del Target CPU



EJEMPLO No. 32

Uso básico del “Processor Expert™”

Objetivo:

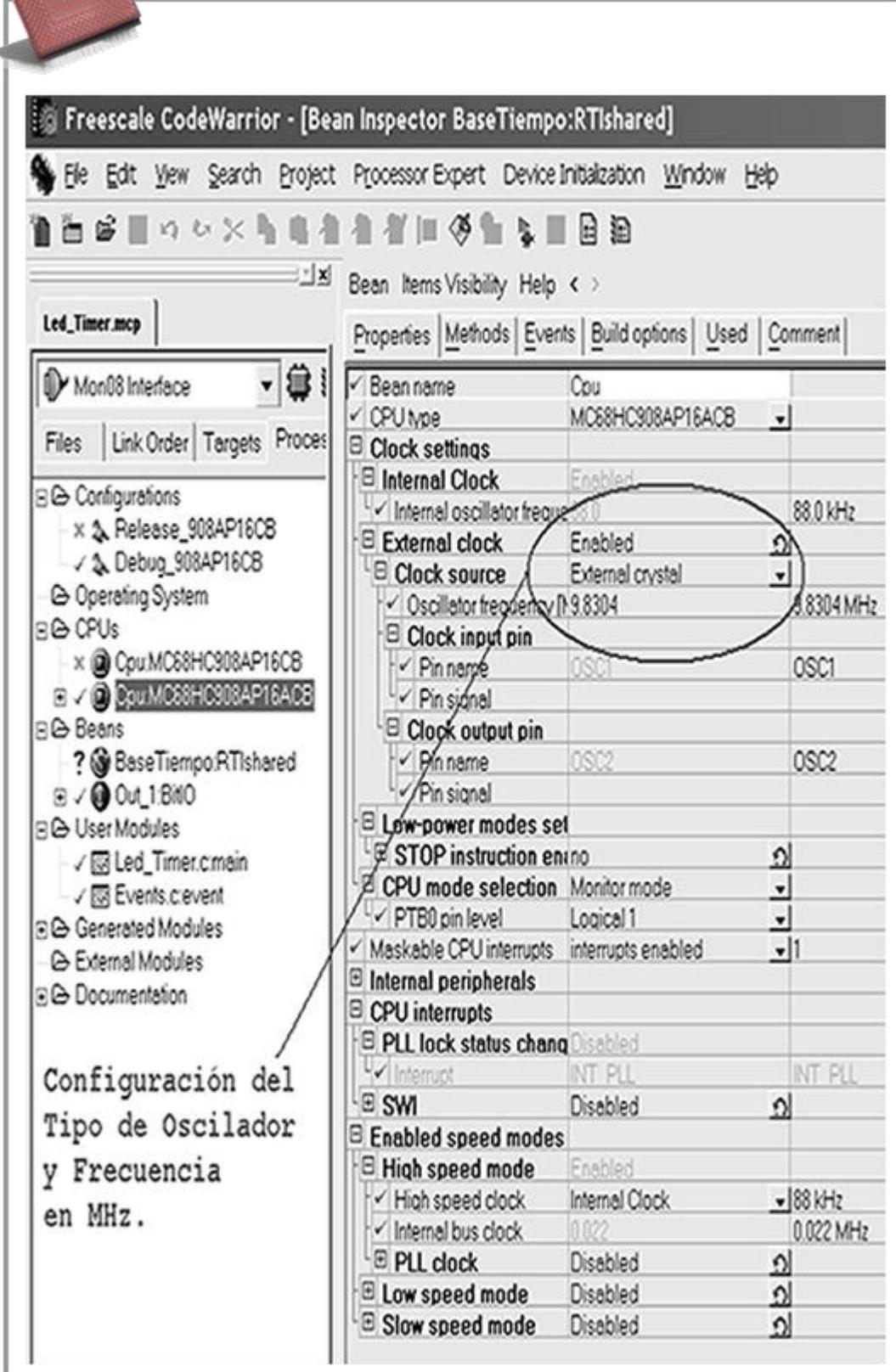
Utilizar el “*Processor Expert™*” para crear una base de tiempo de un (1) segundo, usando alguno de los periféricos internos de manejo de tiempo del microcontrolador AP16. Verificar el correcto funcionamiento mediante la salida OUT-1 del sistema AP-Link.

Solución:

Inicialmente se debe configurar el “*bean*” CPU con el tipo de oscilador que usará el sistema. En este caso particular se usa un oscilador externo (X1 del sistema AP-Link), que tiene una frecuencia de 9.8304MHz (Gráfico 9.8).

**GRÁFICO
9.8**

Selección del tipo de oscilador y frecuencia



Se tendrán en el “**bean**” CPU dos métodos básicos:

EnableInt: creará la función **void Cpu_EnableInt(void);** está permitirá habilitar las interrupciones al ser invocada.

DisableInt: creará la función **void Cpu_DisableInt(void);** esta permitirá deshabilitar las interrupciones a la CPU una vez sea invocada por el programador.

Estas funciones son útiles en cualquier proyecto para el manejo de las Zonas Críticas de software,y es recomendable generar su código respectivo (ver *Gráfico 9.9 Methods*).

GRÁFICO
9.9

Methods: métodos del "bean" CPU.

Freescale CodeWarrior - [Bean Inspector BaseTiempo:RTIshared]

File Edit View Search Project Processor Expert Device Initialization Window Help

Bean Items Visibility Help < >

Led_Timer.mcp

Mon08 Interface

Properties Methods Events Build options Used Code

SetHighSpeed	don't generate code
SetLowSpeed	don't generate code
SetSlowSpeed	don't generate code
GetSpeedMode	don't generate code
SetIntVect	don't generate code
GetIntVect	don't generate code
EnableInt	generate code
DisableInt	generate code
GetResetSource	don't generate code
SetWaitMode	don't generate code
SetStopMode	don't generate code
Delay100US	don't generate code
GetLowVoltageFlag	don't generate code

Configurations

- Release_908AP16CB
- Debug_908AP16CB

Operating System

CPUs

- Cpu:MC68HC908AP16CB
- Cpu:MC68HC908AP16ACB

Beans

- BaseTiempo:RTIshared
- Out_1:BitIO

User Modules

- Led_Timer.c:main
- Events.c:event

Generated Modules

External Modules

Documentation

Habilitar generación de Código para estas 2 funciones

The screenshot shows the Freescale CodeWarrior interface with the 'Bean Inspector' window open. The title bar reads 'Freescale CodeWarrior - [Bean Inspector BaseTiempo:RTIshared]'. The menu bar includes File, Edit, View, Search, Project, Processor Expert, Device Initialization, Window, and Help. Below the menu is a toolbar with various icons. The main area has tabs for Bean, Items, Visibility, Help, and navigation arrows. A sub-menu bar shows 'Properties', 'Methods' (which is selected), 'Events', 'Build options', 'Used', and 'Code'. On the left is a tree view of the project structure under 'Led_Timer.mcp': Mon08 Interface, Configurations (Release_908AP16CB, Debug_908AP16CB), Operating System, CPUs (Cpu:MC68HC908AP16CB, Cpu:MC68HC908AP16ACB), Beans (BaseTiempo:RTIshared, Out_1:BitIO), User Modules (Led_Timer.c:main, Events.c:event), Generated Modules, External Modules, and Documentation. The 'Methods' tab is active, displaying a table of methods for the 'BaseTiempo:RTIshared' bean. The table has two columns: method name and code generation status. Methods listed include SetHighSpeed, SetLowSpeed, SetSlowSpeed, GetSpeedMode, SetIntVect, GetIntVect, EnableInt, DisableInt, GetResetSource, SetWaitMode, SetStopMode, Delay100US, and GetLowVoltageFlag. Most methods have 'don't generate code' in the status column, except for 'EnableInt' and 'DisableInt' which have 'generate code'. A large oval highlights the 'EnableInt' and 'DisableInt' rows. A callout line points from the text 'Habilitar generación de Código para estas 2 funciones' at the bottom left to the highlighted row in the table.

folder Events es posible habilitar funciones que permiten ejecutar algún código cuando se presenta un evento particular, como puede ser:

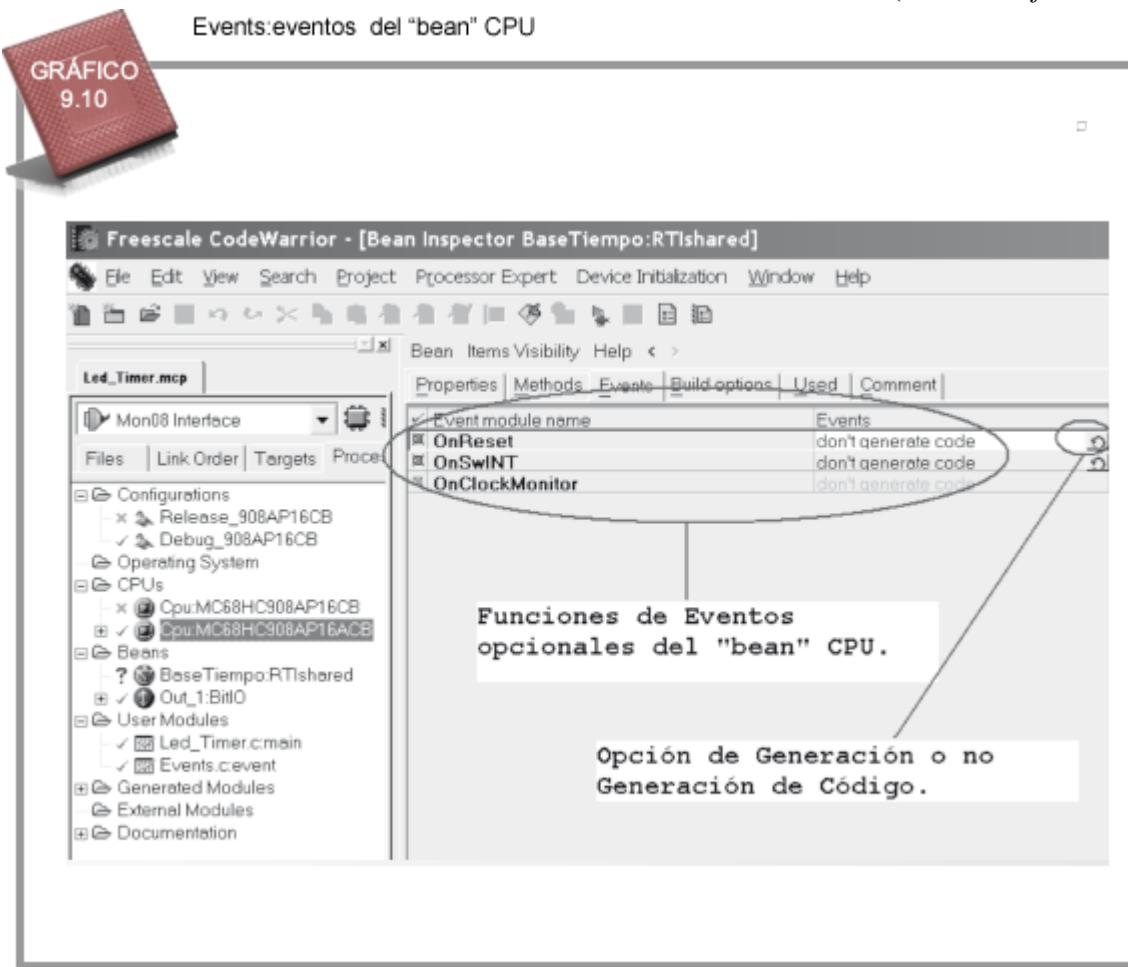
OnReset: habilita la generación de la función **void Cpu_OnReset(void);** que permite que un código sea ejecutado cuando se presenta un **Reset** en la máquina.

OnSwiINT: habilita la generación de la función **void Cpu_OnSwiINT(void);** que permite ejecutar algún código cuando se presenta una interrupción de software (SWI).

OnClockMonitor: si es habilitada, genera la función **void Cpu_OnClockMonitor(void);** con el cual el programador ejecuta algún código cuando el monitor del PLL genera una interrupción.

Para la solución de este ejemplo estas funciones no serán requeridas y con el objetivo de ahorrar espacio en memoria, serán deshabilitadas, esto se realiza seleccionando la opción “don’t generate code” en la columna “Events” del “bean” CPU (ver Gráfico 9.10 Events).

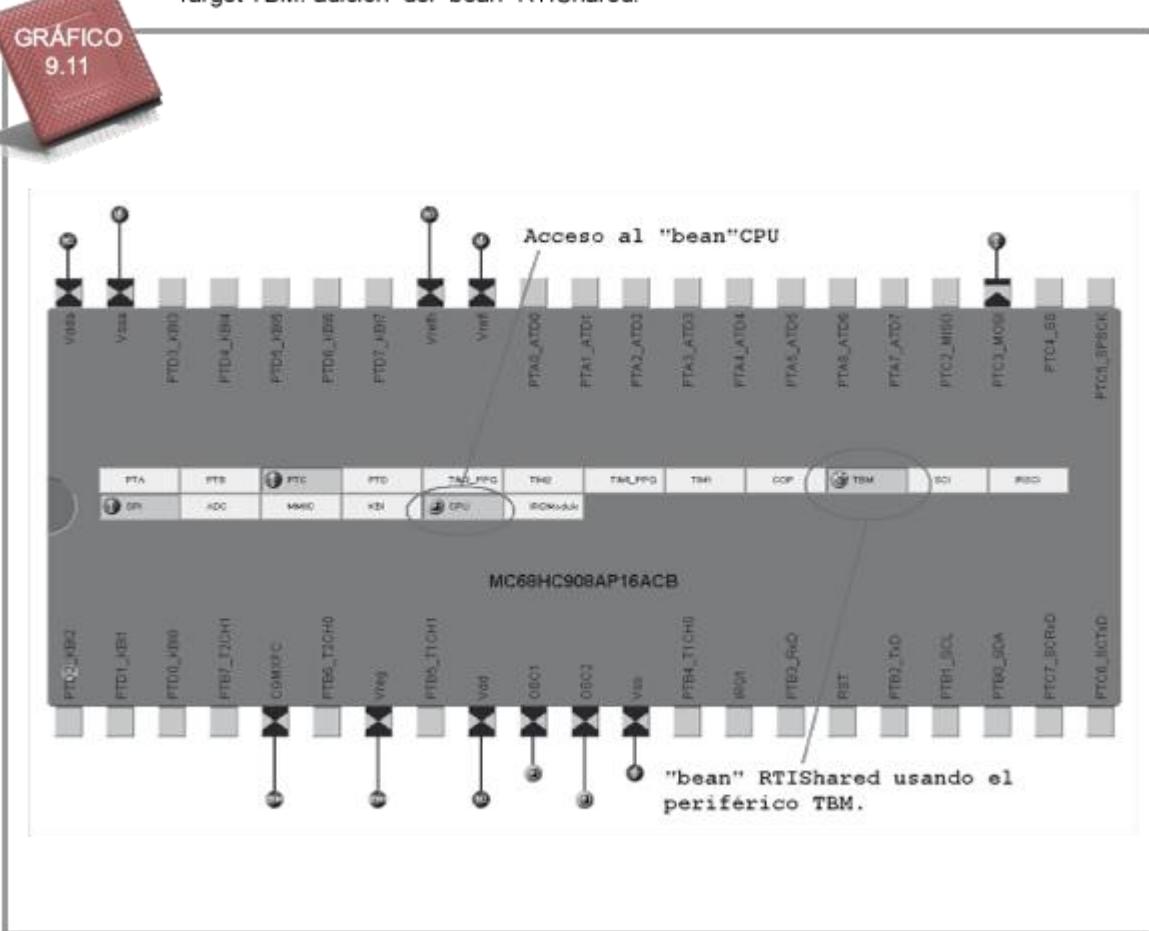
Events: eventos del “bean” CPU



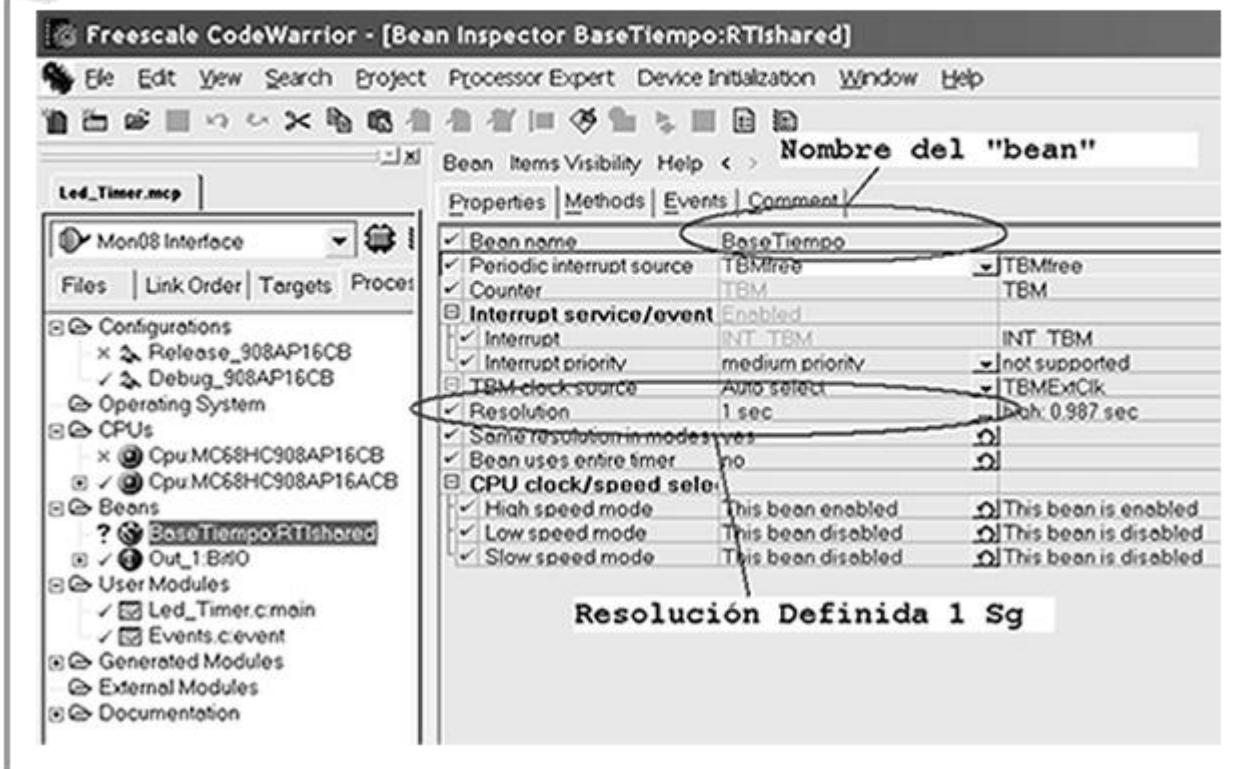
Si el programador en algún momento desea realizar alguna acción ante los eventos mencionados, podrá luego habilitarlos y el “Processor Expert™” generará estas funciones en el código en C. La configuración básica del “bean” CPU estará terminada y la ventana puede ser minimizada. Se selecciona ahora uno de los periféricos de base de tiempo **TBM** (Time Base Module) en la ventana **Target CPU**. Si los periféricos no aparecen en esta ventana, puede expandir o maximizarla para tener acceso a ellos.

Mediante el botón derecho del mouse se adiciona un nuevo “bean”, en modo **RTIShared**.

Target TBM: adición del “bean” RTIShared.



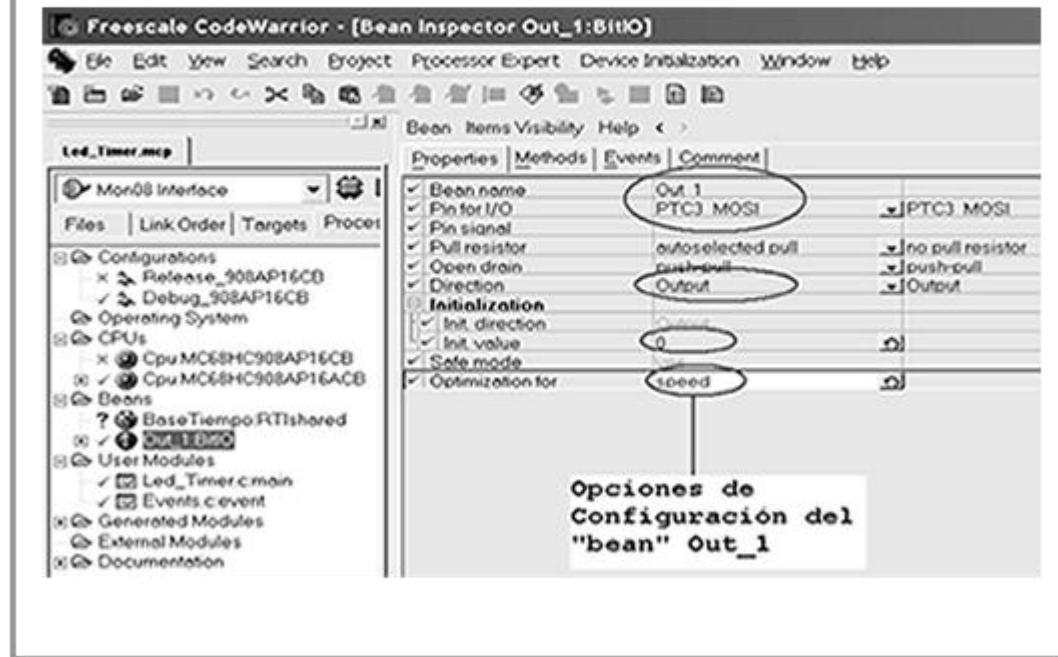
Se accesa luego al folder **“Properties”**, se modifican los campos: **“Bean name”**: con el nombre que se le dará al “bean” (ejemplo **BaseTiempo**). **“Resolution”**: se selecciona un (1) segundo como período de la interrupción de **TBM**. En los demás folders no existen métodos ni eventos por habilitar,dado que este “bean” funciona con base en interrupciones.

GRÁFICO
9.12

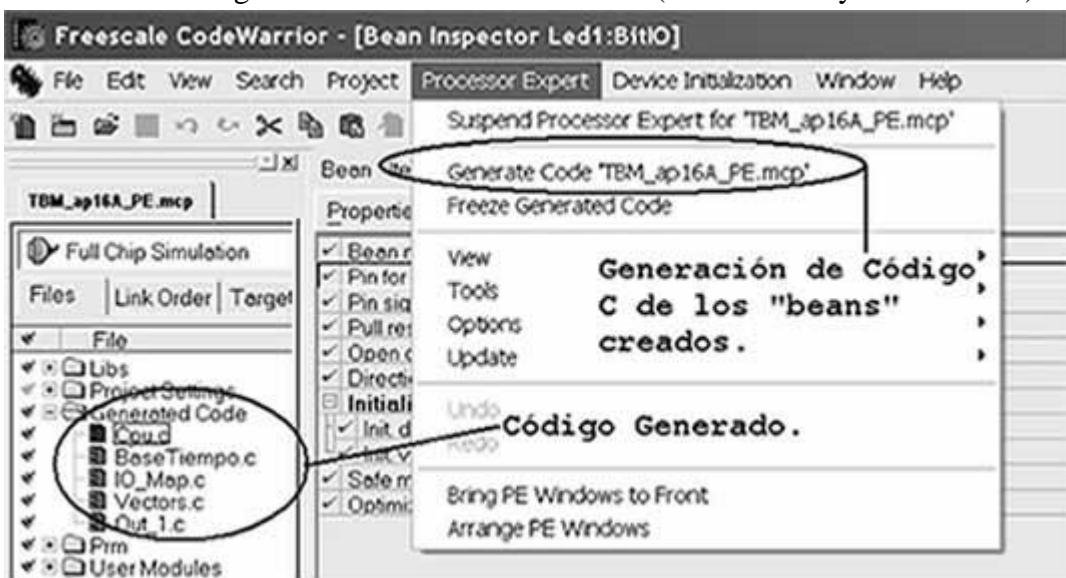
Se crea un nuevo “**bean**” para el manejo del Led OUT-1, ubicado en el **PTC 3** del microcontrolador, mediante el acceso a la ventana Target CPU → Periférico **PTC**, se crea un nuevo “**bean**” de tipo **BitIO**, al accesar sus propiedades se define: “**Bean name**”: con el nombre que se le dará a ese “**bean**” → Out_1. “**Pin for I/O**”: se selecciona el pin PTC 3 al cual está conectado el led OUT-1 del AP-Link. “**Direction**”: se selecciona Output (pin es de Salida). “**Init. value**”: se selecciona 0 (el estado inicial del Led OUT-1 es apagado). “**Optimization for**”: puede seleccionarse que optimice el código por velocidad (speed) en el cual se generarán más macros (#define), o bien, seleccionar densidad de código (code size) cuando la velocidad de ejecución de las funciones a generar no es crítica y se requiere que la capacidad de memoria que ocupa sea menor.



RTIShared: propiedades del "bean" Out_1.



En la ventana de “Methods” se habilitan 3 de ellos, que serán los que se usarán: ClrVal → Apaga la salida Out_1. SetVal → Enciende la salida Out_1. NegVal → Invierte el estado de la salida Out_1 (Función Toggle), será la función usada en este Una vez realizada la configuración se accede al menú Processor Expert → Generate Code, que indicará posibles errores en las configuraciones de los “beans”, y creará los códigos fuentes (.H) y respectivos (.C).

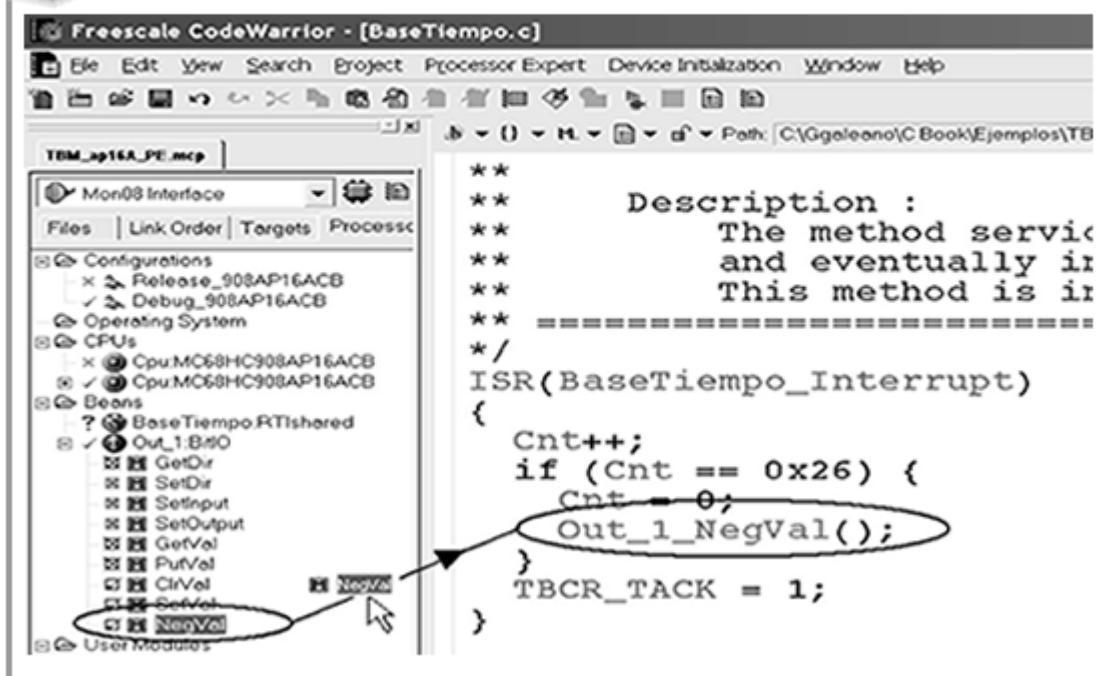


Terminada la creación y configuración de los “beans” que requiere la solución del enunciado. Se abre el archivo **Base_Tiempo.c** y se accede a la función de interrupción **ISR(BaseTiempo_Interrupt)**, y se arrastra

(drag&drop) la función NegVal de Out_1, llamada **Out_1_NegVal()**;

(drag & drop) Arrastre de funciones creadas por "processor Expert"

GRÁFICO
9.14



Y se incluye

el archivo de cabecera **#include “Out_1.h”** ya que se usará una función de este módulo. Se compila el programa, presionando F7 y se envía a la tarjeta AP-Link con F5. Como resultado el Led Out_1 en la tarjeta AP-Link deberá encender y apagar cada segundo. **Discusión:** El ejemplo desarrollado muestra paso a paso cada una de las ventanas básicas de un proyecto realizado usando la herramienta “Processor Expert™” para generar el

código en lenguaje abierto C. Se generaron para este proyecto particular 2 “beans”, uno para el manejo del tiempo llamado **BaseTiempo** y **Out_1**, implícitamente al proyecto está el “bean” CPU que es obligatorio configurar en cualquier proyecto por contener parámetros propios de la máquina usada. Al acceder a estos “beans” mediante el botón derecho del mouse, se puede configurar sus propiedades, métodos y eventos, cuidando de solo crear el código necesario con el fin de economizar recursos del procesador. Una vez generado el código básico de cada componente, se procede a modificar los archivos fuentes que cumplen con la especificación del proyecto. El resultado, mayor velocidad de desarrollo sin necesidad de conocer a detalle el procesador particular usado.

9.6 CREACIÓN DE BEANS EN “PROCESSOR EXPERT™”

Materiales adicionales en la



Uso básico del processor expert:
simulación y proyecto completo para
Freescale™ (AP-Link).



Los **beans** pueden ser almacenados como plantillas que luego son llamadas e integradas a nuevos proyectos.

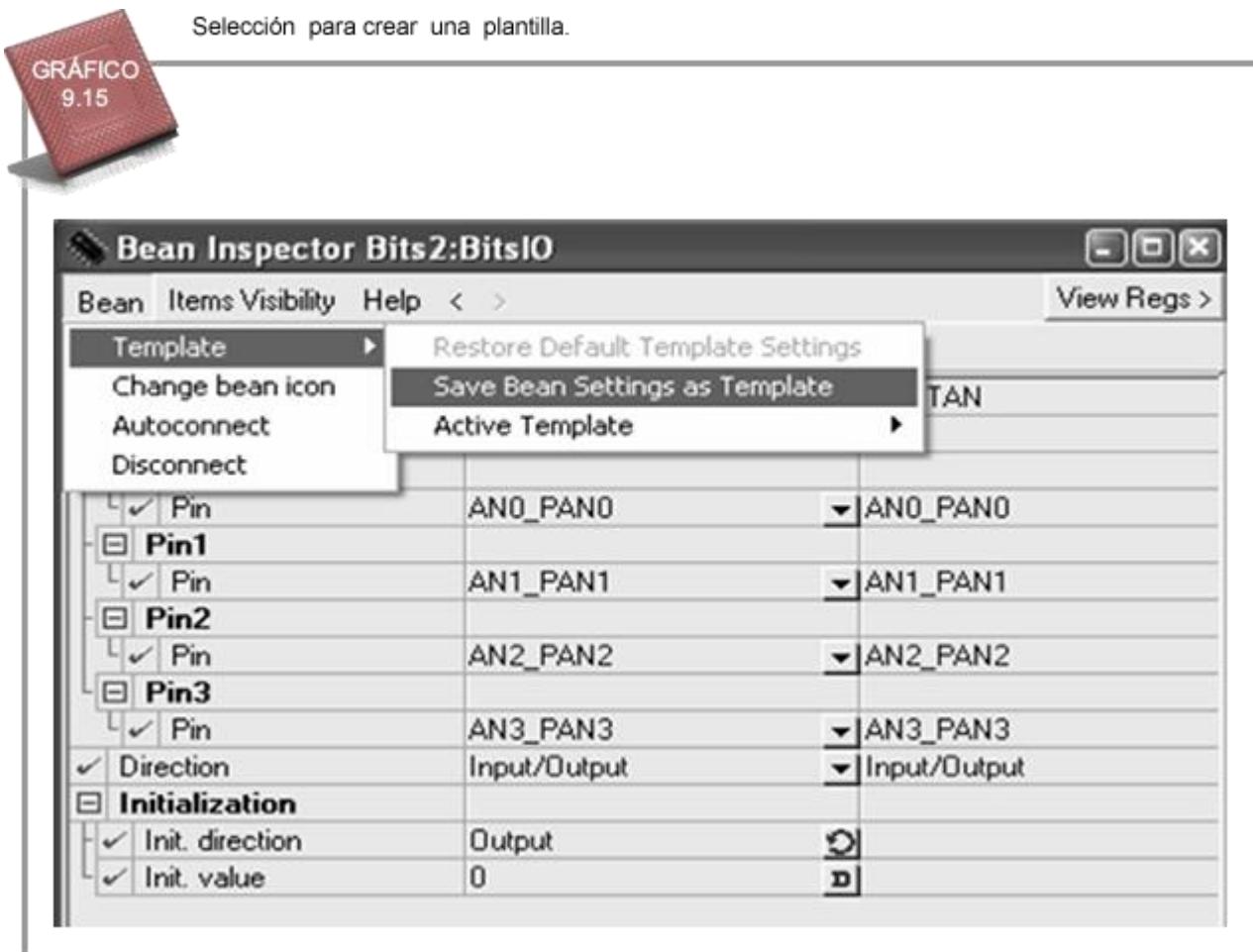


Además del uso de las librerías propias del Codewarrior®, la herramienta tiene la infraestructura que permite la creación de *beans* propios, de tal forma que un módulo de software puede ser encapsulado usando el “**Bean Wizard**”, quedando embebido dentro del IDE del Codewarrior® para uso posterior.

9.6.1 Creación de un bean plantilla

Un bean con determinada configuración puede ser almacenado como una plantilla (bean template), que luego puede ser invocado y agregado al mismo o a otros proyectos.

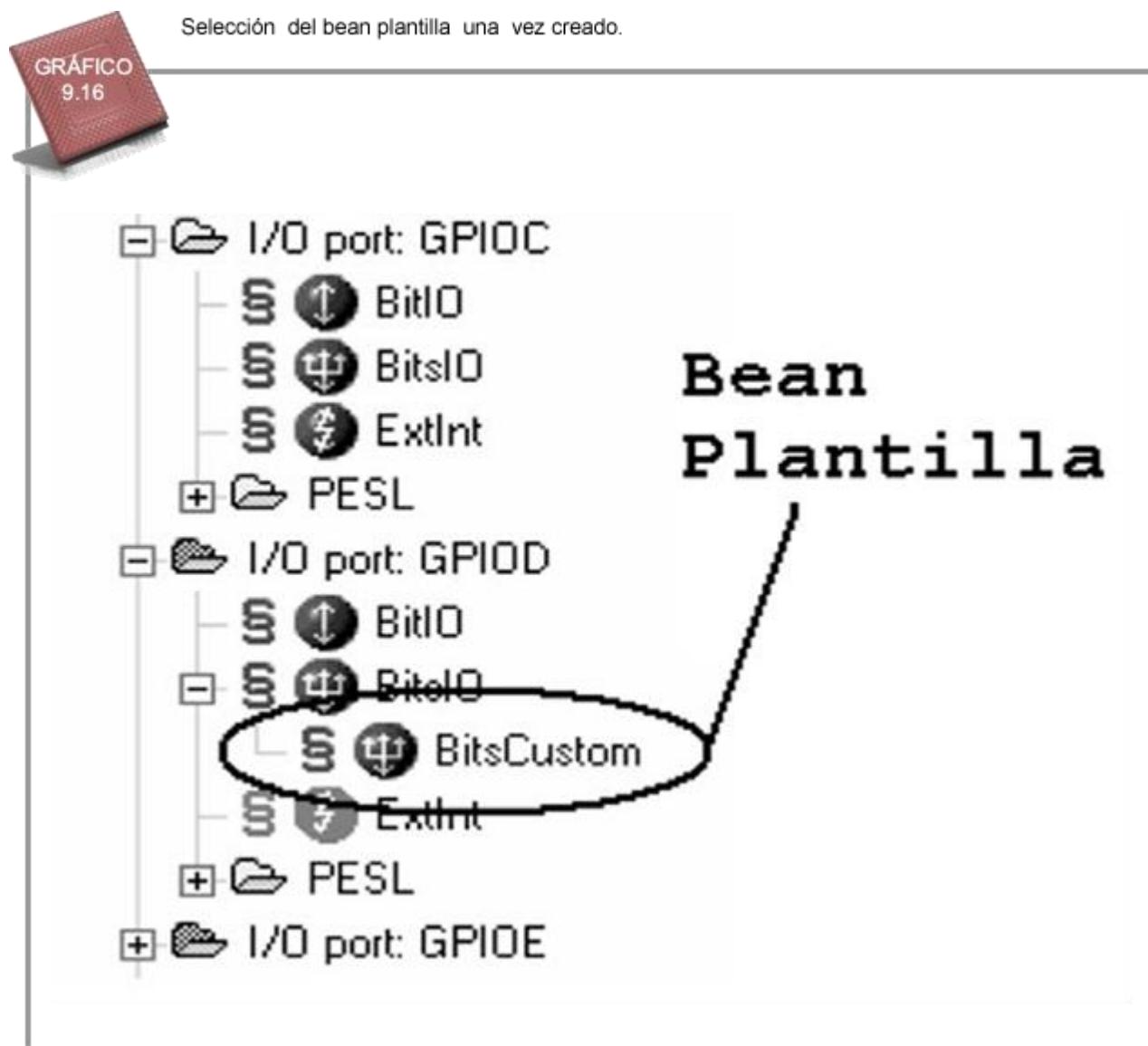
Esta facilidad está disponible en el menú **Bean → Template → Save Bean Settings as Template** (*ver Gráfico 9.15*).



Se define el nombre que se le dará a la plantilla, y el periférico asociado que usará.

La siguiente y última ventana **TEMPLATE EDITOR**, permitirá definir la configuración de la plantilla, además, facilita definir algunos atributos como de solo lectura y el método de generación de código y de eventos.

Una vez creada la plantilla es visualizada en la ventana bean selector junto a los beans originales, con la diferencia que el bean plantilla se presenta es coloreado de azul.



9.6.2 Uso del *Bean Wizard*™



Con la tecnología

de los beans el diseñador puede encapsular toda una sección de código optimizado e incorporarlo a otras plataformas o compartirlo con miembros del equipo de trabajo.



Un código en lenguaje C ensayado y optimizado, puede encapsularse y convertirse en un bean, que a su vez puede ser usado en nuevos proyectos que requieran incorporarlo, o bien portarlo, a otras tecnologías y plataformas.

La forma que Processor Expert™ sugiere para este tipo de código común es convertir esta sección en un código embebido reutilizable (embedded bean), con las siguientes ventajas:

Simplifica la generación del código final.

Adiciona claridad a un proyecto.

Evita errores en transcripción disminuyendo los riesgos de defectos.

Permite que el hardware sea accesado a través de capas superiores (HAL).

Facilita la migración entre plataformas.

El **Bean Wizard™** es una parte del **Processor Expert™**, que permite crear beans propietarios y facilita a los diseñadores encapsular un código en C en componentes de fácil uso y distribución a un grupo de trabajo.

La configuración del nuevo *bean* se establece con base en la siguiente tabla:

Relación entre aplicación y componente del bean.	
Pregunta en capa de aplicación	En términos de beans significa...
¿Qué debe el usuario ver y usar?	Definición del API.
¿Qué funciones de acceso tendrá el usuario?	Definición de métodos (<i>methods</i>).
¿Qué hace cada función?	Funcionalidad del método.
¿Qué respuesta espera el usuario del bean?	Eventos (<i>events</i>).
¿Requiere acceso a periféricos o solo es exclusivamente software?	Conexión con otros beans.
¿Existen macros (<i>defines</i>)?	
¿Existe algo que modifique la funcionalidad?	Propiedades.

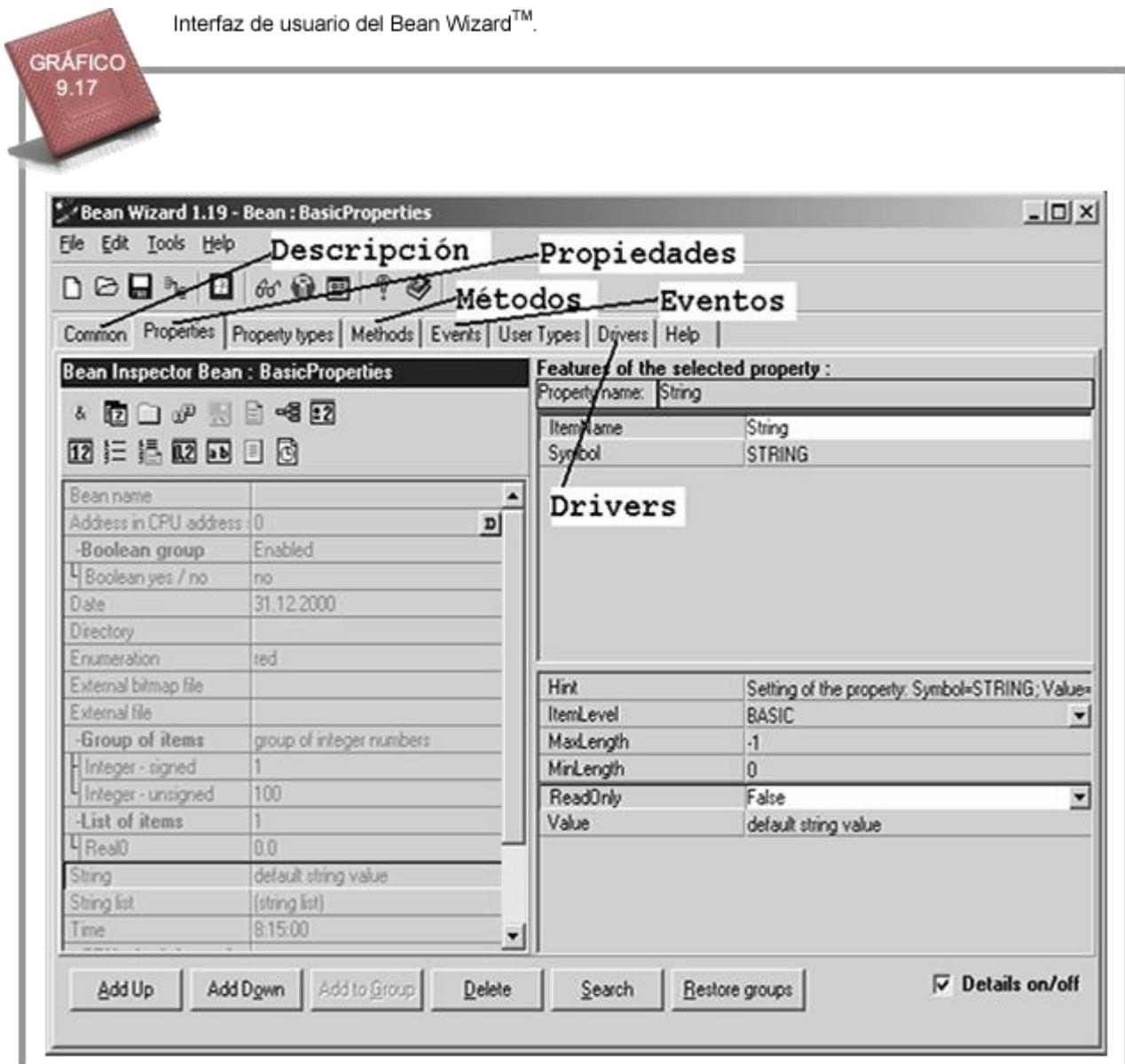
El proceso de creación se realiza con el acceso a cada uno de los tabs de la ventana de interfaz de usuario:

Common, especifica la descripción del componente.

Properties, se define el nombre, el tipo, intrucciones de ayuda (hints) y los símbolos del bean.
Methods, permite definir nombre de los métodos, parámetros y tipos de retorno con las respectivas instrucciones de ayuda (hints).

Events, acceso a la definición del nombre de los eventos con los respectivos parámetros.

Drivers, permite la definición de los drivers para la generación del código: métodos, plantillas de eventos, código de inicialización y los archivos de cabecera (includes). Permite además la conexión con otros beans.



Para la creación de los beans, en términos generales, se sugiere llevar a cabo los siguientes pasos:

Análisis: identificar en el código las funciones públicas, las funciones internas o privadas, el comportamiento de los modificadores, necesidad de incluir archivos de cabecera, la dependencia con la inicialización del bean CPU.

Definición de la interfaz: definir las propiedades de lo que el usuario puede cambiar o definir al momento del diseño. Estas propiedades modifican el código generado. La definición de los métodos, que son funciones a las que el usuario accede, requiere ser implementada así como los eventos que se generan al exterior.

Implementación del Driver: definir la implementación de los métodos, los archivos de cabecera y las variables globales. Inicializar el *bean*, definir archivos de chequeo, cambios y prueba al momento del diseño.

Prueba: realización de varios proyectos usando el **Processor Expert™**, para la prueba completa del *bean*.

Integración y distribución: un nuevo bean estará visible y se podrá usar cada que se realice un proyecto usando el *Processor Expert™*.

EJEMPLO No. 33

creación básica de beans

Objetivo:

Con la ayuda del Bean Wizard™ de Processor Expert™, crear un bean reutilizable que entregue el bit de verificación de paridad de un byte.

Solución:

Este bean tendrá un solo método GetParityBit, y una propiedad que es el tipo de paridad que se requiere: par (even) o impar (odd).

El cálculo de la paridad de un byte(b7...b0) se realiza por medio de la función lógica XOR de susbits:

b0^b1^b2^b3^b4^b5^b6^b7

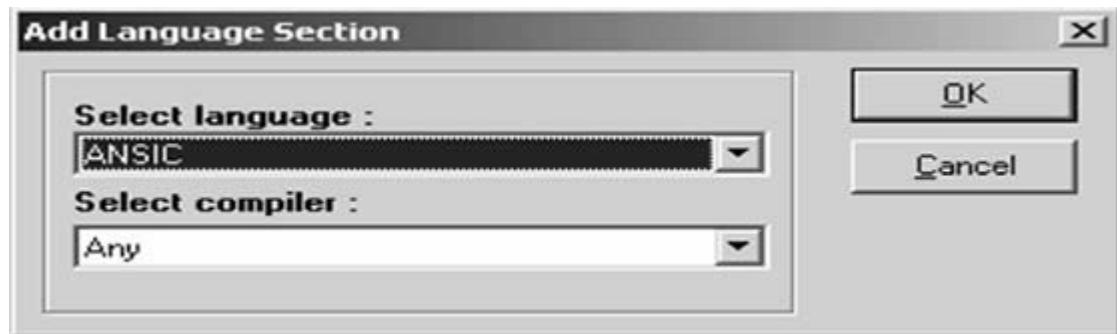
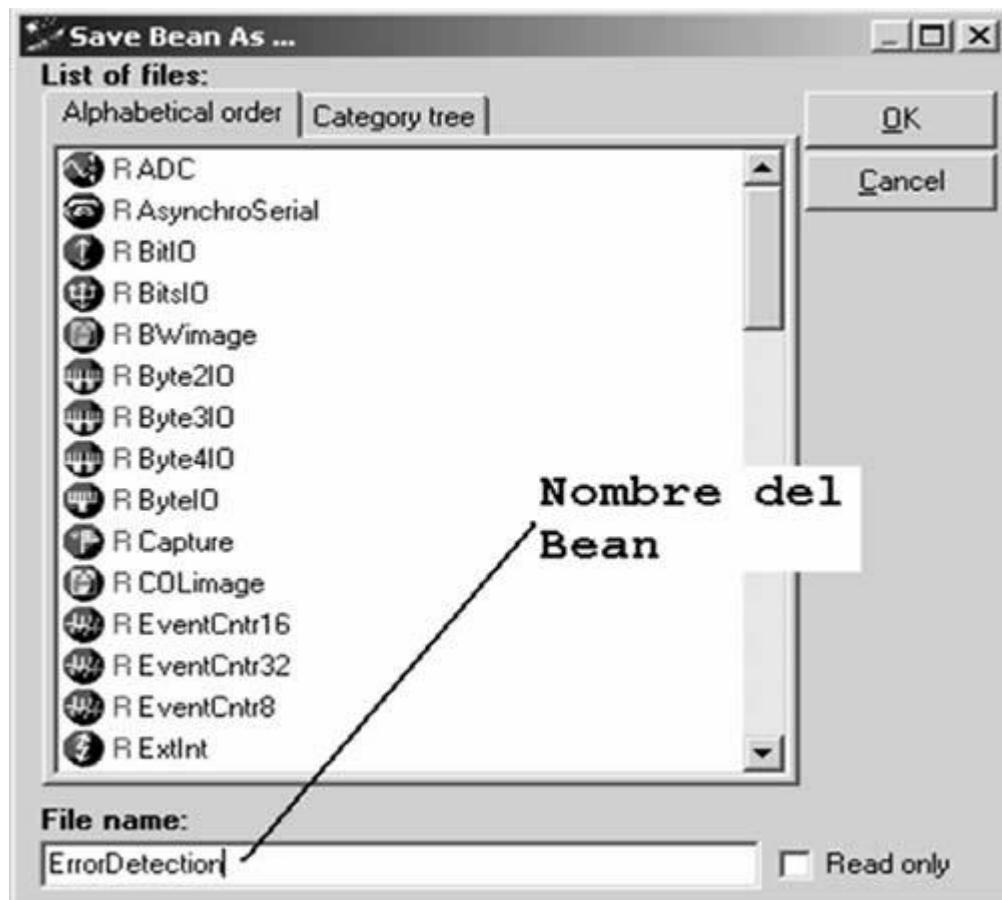
Para crear el bean correspondiente se realizan los siguientes pasos:

Iniciar el IDE de Codewarrior® y seleccionar el **Processor Expert → Tools → Bean Wizard menu**



En la ventana **Bean Wizard** → **Common**, se ingresa la descripción, se definen las propiedades, y los métodos.

Se guarda con **Save Bean As...** en este caso como **ErrorDetection**, y se selecciona “Yes” para crear un nuevo driver de software en lenguaje ANSI C.



Se definen las propiedades del bean, la cual en este caso el usuario deberá cambiar entre *odd* o *even*:

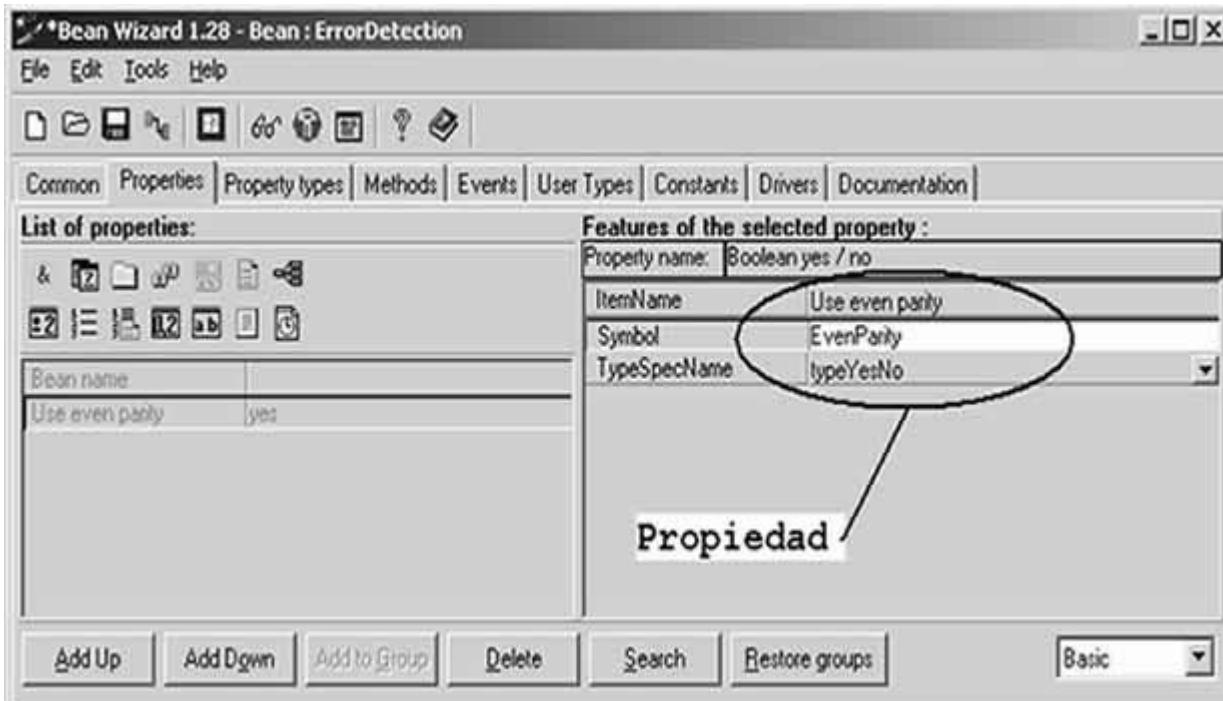
Se presiona el botón “**Add Down**” seleccionar “**Boolean Yes/No**” y presionar **OK**.

Llenar los espacios así:

Item Name, digitar **Use Even Parity**.

Enter Symbol, digitar **EvenParity**.

Type spec name, digitar **typeYesNo**.



En el tab de métodos:

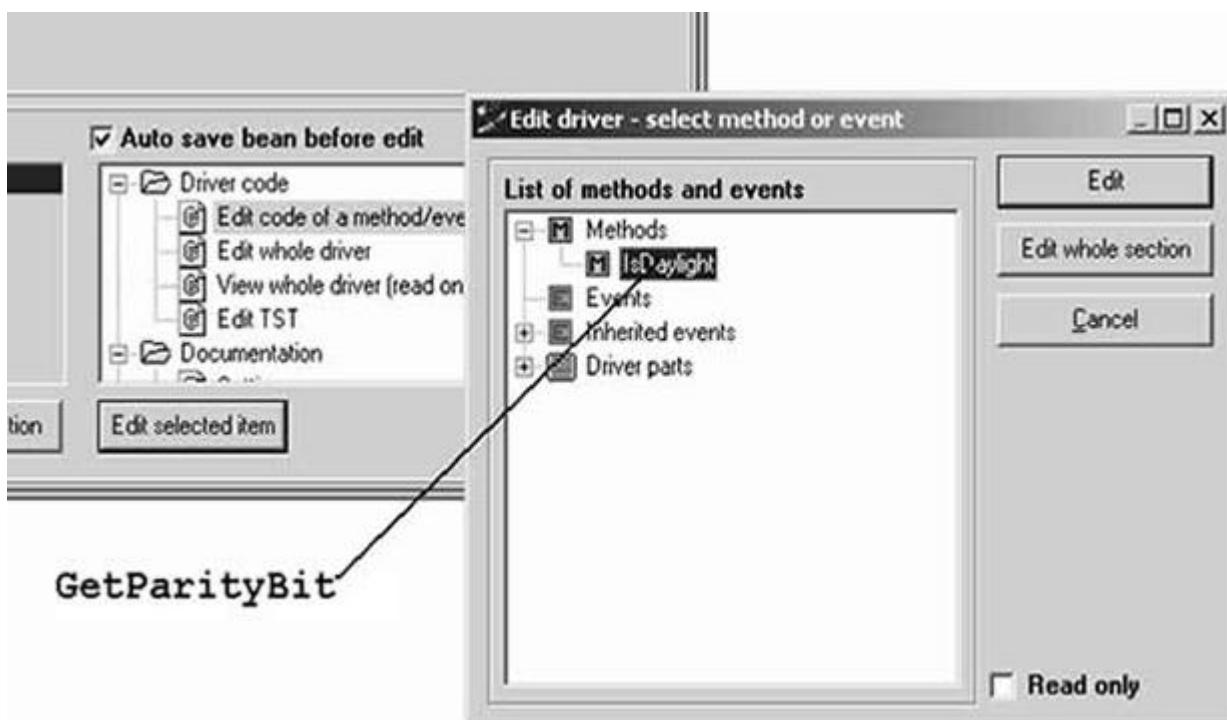
Presionar el botón “Add”, se entra en nombre del método **GetParityBit**, return type **8bit unsigned**.

Presionar en botón de “Add parameter”, y digitar **Data**, con tipo **8bit unsigned**.

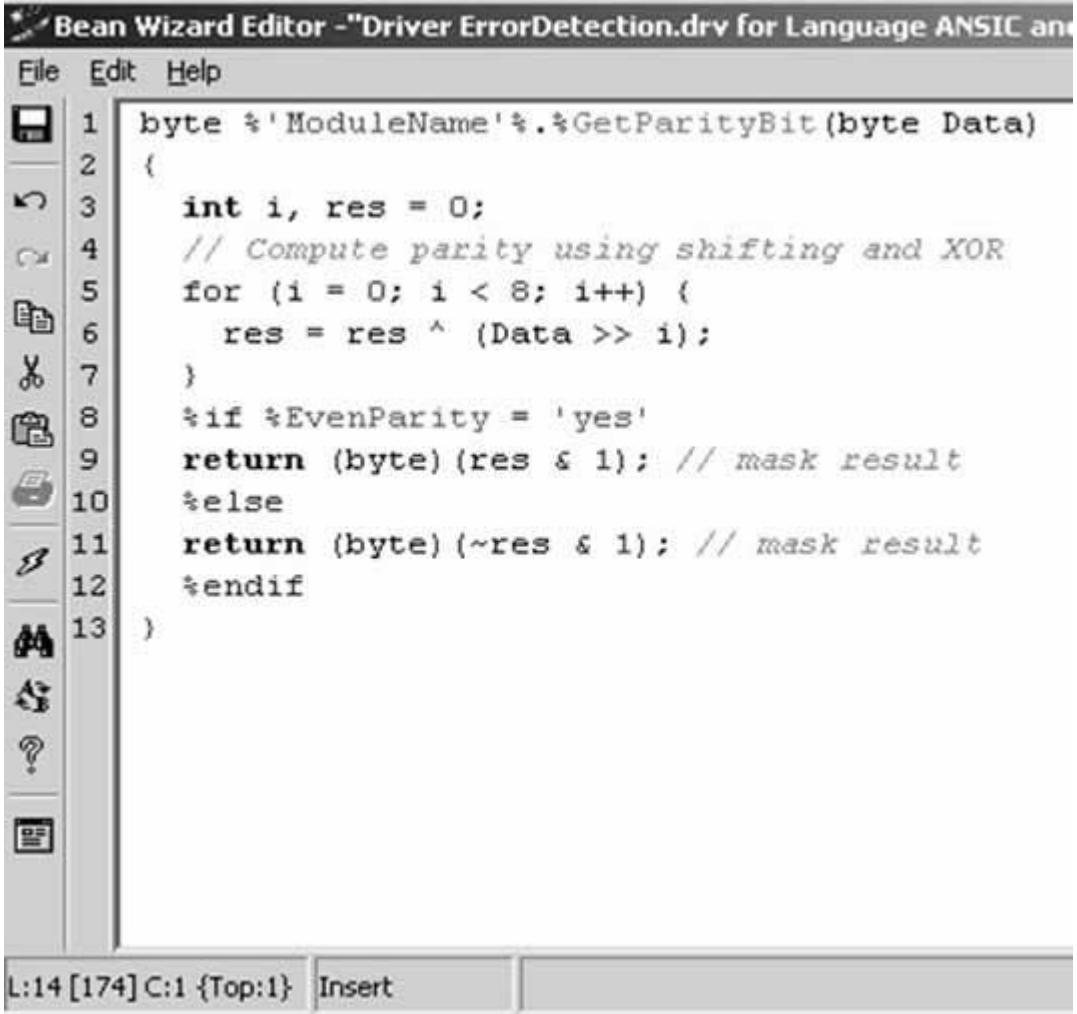


En esta parte se concluye la definición del interface del bean.

Para la implementación del **driver**, presionar en **Drivers**, y seleccionar **Edit code of method/event** y botón **Edit selected item**, seleccionar **getParityBit → Edit**.



Se inserta el código:



The screenshot shows the Bean Wizard Editor interface with the title "Bean Wizard Editor -"Driver ErrorDetection.drv for Language ANSIC and C" at the top. The menu bar includes File, Edit, and Help. On the left is a toolbar with various icons. The main window displays the following C code:

```
byte %'ModuleName'%.%GetParityBit(byte Data)
{
    int i, res = 0;
    // Compute parity using shifting and XOR
    for (i = 0; i < 8; i++) {
        res = res ^ (Data >> i);
    }
    %if %EvenParity = 'yes'
    return (byte) (res & 1); // mask result
    %else
    return (byte) (~res & 1); // mask result
    %endif
}
```

The status bar at the bottom shows "L:14 [174] C:1 {Top:1} Insert".

Discusión:

Con la facilidad que provee el Bean Wizard™, es posible crear componentes de software que pueden ser incluidos en proyectos creados con el Processor Expert™, facilitando de esta forma que capas superiores de aplicación usen un código ya probado y

muy portable.

En el ejemplo se ilustra un caso muy sencillo de creación de un bean, de un solo método y una sola propiedad por configurar, de forma similar se pueden replicar el ejemplo y crear componentes de mediana complejidad.

El bean elaborado es exclusivamente de software operacional y se creó siguiendo los pasos recomendados para la creación de beans, que permite de forma sistemática llegar a la solución final del nuevo bean. De la misma forma que se prueba una aplicación desarrollada en C, este proyecto final debe ser ensayado realizando varios proyectos con distintos valores que prueben las propiedades del código generado y su función al tiempo que se genera un código óptimo y confiable; en muchos casos es bueno acudir al ensayo y error hasta lograr sintonizar el componente y garantizar el funcionamiento.

La sintaxis en la implementación del Driver no es 100% ANSI C, corresponde a un lenguaje de mayor nivel denominado el Macro Lenguaje, que no es parte del contenido de este libro, sin embargo, en la página web se puede acceder a documentación específica sobre este tema y sobre la creación de beans mas complejos.

9.7 CONSIDERACIONES SOBRE EL USO DEL “PROCESSOR EXPERT™”



Para el proyecto del generado que código sea de manera

óptima, es importante iniciar el trabajo definitivo en C a fin de tener total control sobre el código.



Una vez que los “beans” son configurados, y el código es generado por el “Processor Expert™”, se generan funciones que posiblemente no se usarán en el proyecto final, o que tienen prototipos más amplios que los que el programador consideraría si escribiera el código.

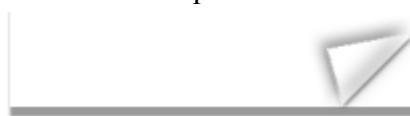
de 8 bits y en muchos casos el programa final no requiere el manejo de prototipos o de retornos.

Por esta razón, deberán ser considerados varios aspectos con el fin de optimizar el tamaño y la velocidad del código generado por el proyecto definitivo que va programado en la máquina.

Iniciar el proyecto definitivo en C: se recomienda iniciar un proyecto para la aplicación final sin usar “Processor Expert™” a fin de tener 100% el control sobre el código generado en el proyecto. Para cada periférico o módulo que se requiere usar del microcontrolador generar un nuevo proyecto aislado en “Processor Expert™” que genere el código para este periférico, y luego de tener las funciones de configuración, llevar al proyecto en C solo las funciones requeridas, haciendo la optimización respectiva de los procedimientos. Así por ejemplo, si se realiza un proyecto que manejará el puerto serial SCI, generar un proyecto en C llamado, **PanelSerial.mcp** (que no contiene “Processor Expert™”), y paralelamente realizar un proyecto temporal en “Processor Expert™”, llamado **PanelSerial_PE.mcp**, una vez sea configurado el “bean” del puerto serial **Modem.c**, se pasan las funciones de inicialización: **Serial_Init()**, las de transmisión serial **Serial_TxData(char *)** y la función de recepción serial **Serial_RxData(char*)**, con sus respectivos macros, archivos de cabecera, funciones de interrupción (ISRs) y funciones de soporte adicionales.



Es aconsejable que el programador trabaje en el modo adecuado a su experiencia (BASIC, ADVANCED o EXPERT), ya que de ésta forma podrá controlar la generación de código que realiza el “Processor Expert™”.



Usar el ícono “don’t generate code”: con el fin de que el “Processor Expert™” genere el mínimo de funciones requeridas en una aplicación, ubicar y deshabilitar las funciones que NO se requieren utilizar en cada uno de los “beans” del proyecto. No todas las funciones sugeridas por el “Processor Expert™” en sus “beans” son requeridas y el programador podrá obviar las que no considere necesarias o que no usará.

Modificaciones del usuario en Events.C: al generar código con el “Processor Expert™”, se crea un archivo .C y uno .H por cada “bean”, se pueden realizar cambios sobre cualquiera de los archivos, sin embargo, si se adiciona un nuevo “bean” o se genera código de nuevo, los cambios no serán mantenidos y el archivo generado será el original de “Processor Expert™”, sin los cambios realizados por el usuario; pero el archivo **Events.C** es alimentado en cada generación de código y los cambios realizados sobre este archivo no se perderán, por esta razón debe intentarse hacer las modificaciones sobre el archivo **Events.C** o directamente sobre las funciones en el folder **FILES ·User Modules**.

Elegir el nivel de experiencia adecuado: en la parte inferior de cada “bean” se tienen 3 botones: BASIC, ADVANCED y EXPERT, elegir entre estos el nivel adecuado con base en la experiencia sobre la máquina o el periférico en particular. En conclusión, no mover los parámetros de la máquina que no se manejan con claridad.

Usar un derivativo amplio: si se usa el “Processor Expert™”, para la preparación de un prototipo rápido o bajas cantidades de producción, se recomienda usar procesadores de media o alta gama, como los **HCS12**, los V1 o superior, con esto el desempeño de la aplicación no se verá afectado por la generación de código extra que se genera y no será necesario invertir mucho tiempo en optimización.



EJEMPLO No. 34

Escritura y lectura de memoria E2PROM

Objetivo:

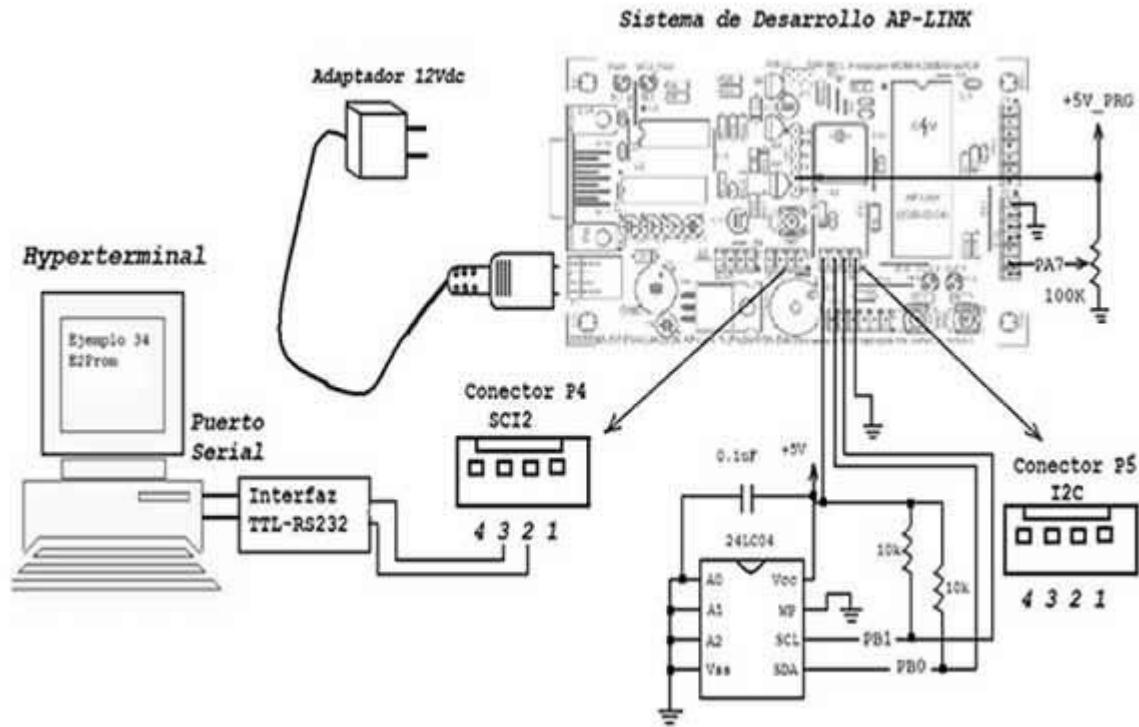
Utilizar el “Processor Expert™” para realizar la escritura en una memoria E2PROM 24LC04, de un valor analógico externo presente en el pin PTA 7/ADC7, cada que sea presionada la tecla INPUT_1, hasta completar 20 valores.

Al presionar la tecla INPUT _2 los valores deberán ser enviados vía serial SCI2 9600 8N1, uno a uno al presionarse la tecla INPUT-1, en formato ASCII de forma que puedan ser visualizados en el hyperterminal de windows. Hacer uso de la máxima resolución del ADC (10 bits) para la adquisición y guardado de los datos.

Verificar que la señal esté dentro del rango entre 2 y 4 voltios, si el valor es menor a 2 voltios, indicarlo mediante encendido y apagado periódico de la salida OUT_1, si la señal tiene un valor superior a 4 voltios, indicarlo con encendido y apagado de la salida OUT_2.

Solución:

Para la prueba real del ejemplo se sugiere el montaje ilustrado a continuación:



Se genera el proyecto en **Codewarrior®** de la forma habitual, con “Processor Expert™” habilitado en el punto 8 de la creación, seleccionando el chip **MC68HC908AP16ACB** en la ventana **Select CPU**.

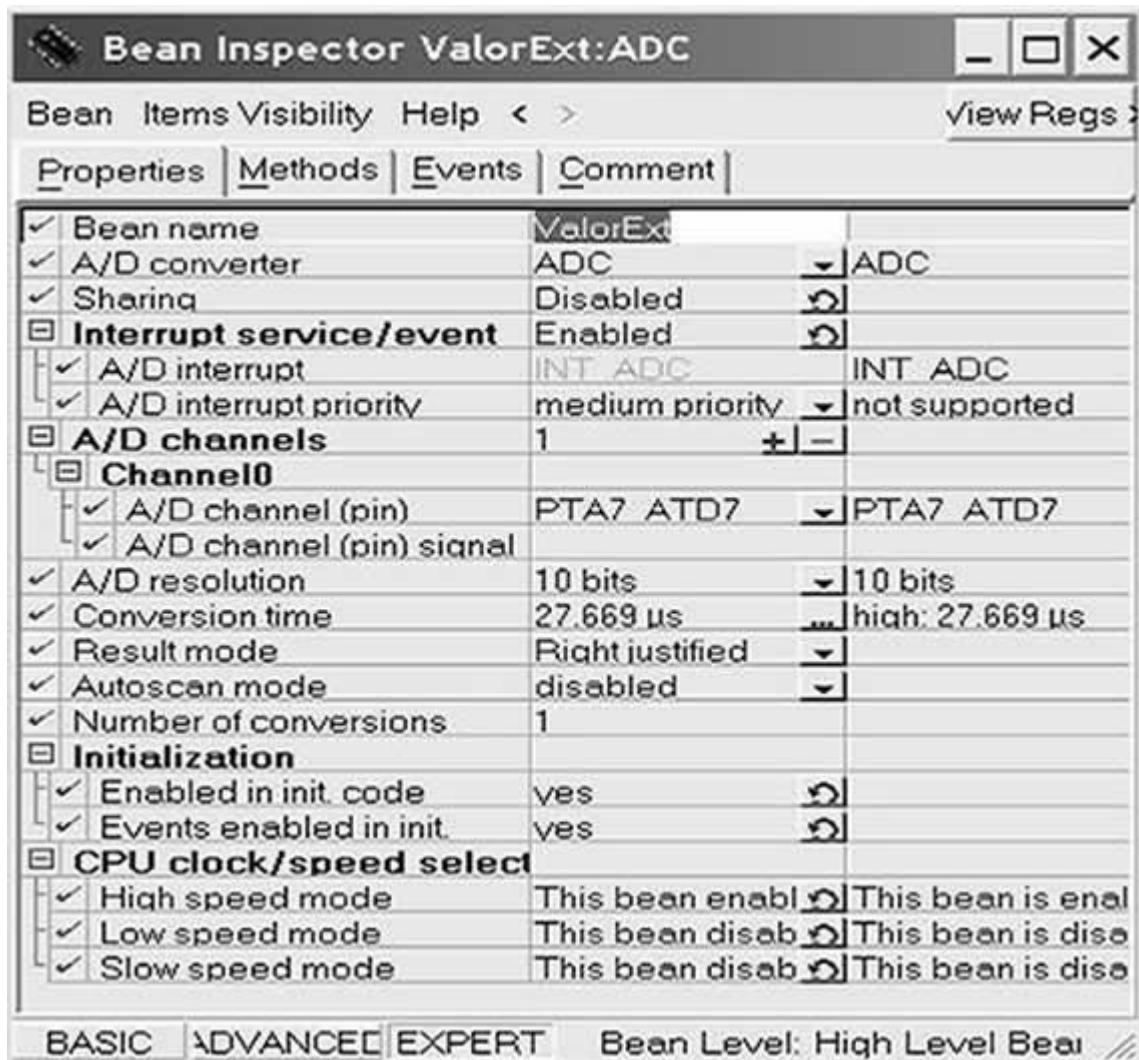
Configuración especial del “bean” CPU:

Exactamente igual al del Ejemplo 32 adicionalmente habilitar los modos de bajo consumo SWI (Enabled) y STOP instruction YES.



Configuración del “bean” ADC:

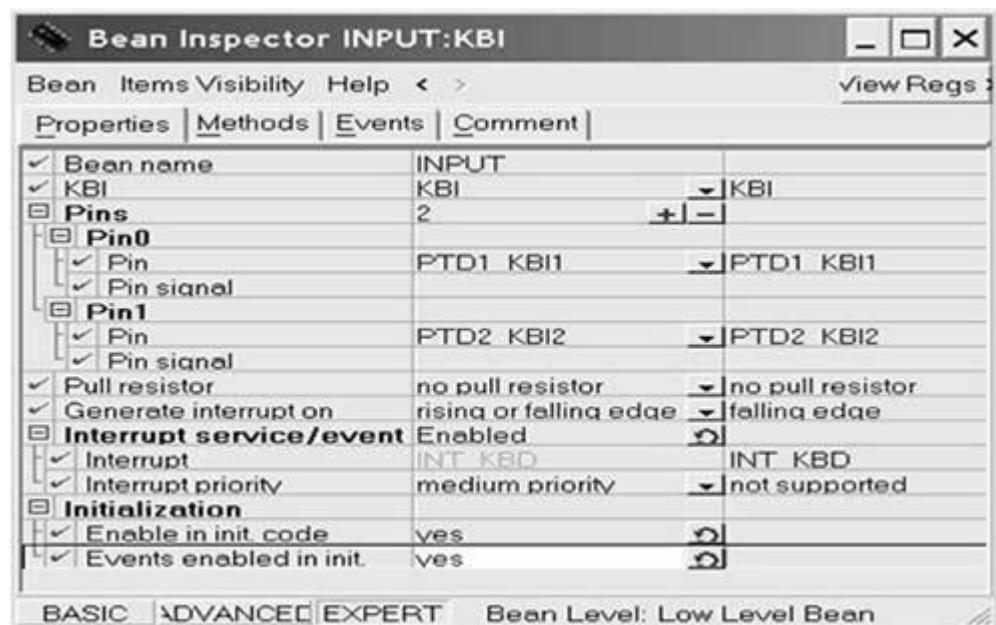
Adicionar un nuevo “bean” ADC que se llamará **ValorExt**, con un canal en PTA 7 ATD 7, quedando así:



En el folder **Methods**, pueden quedar las funciones recomendadas: **Measure()**, y **GetValue1()**, y en el folder **Events** el sugerido **OnEnd()**, el cual genera un evento de llamado a función cada que se termina una conversión en el canal.

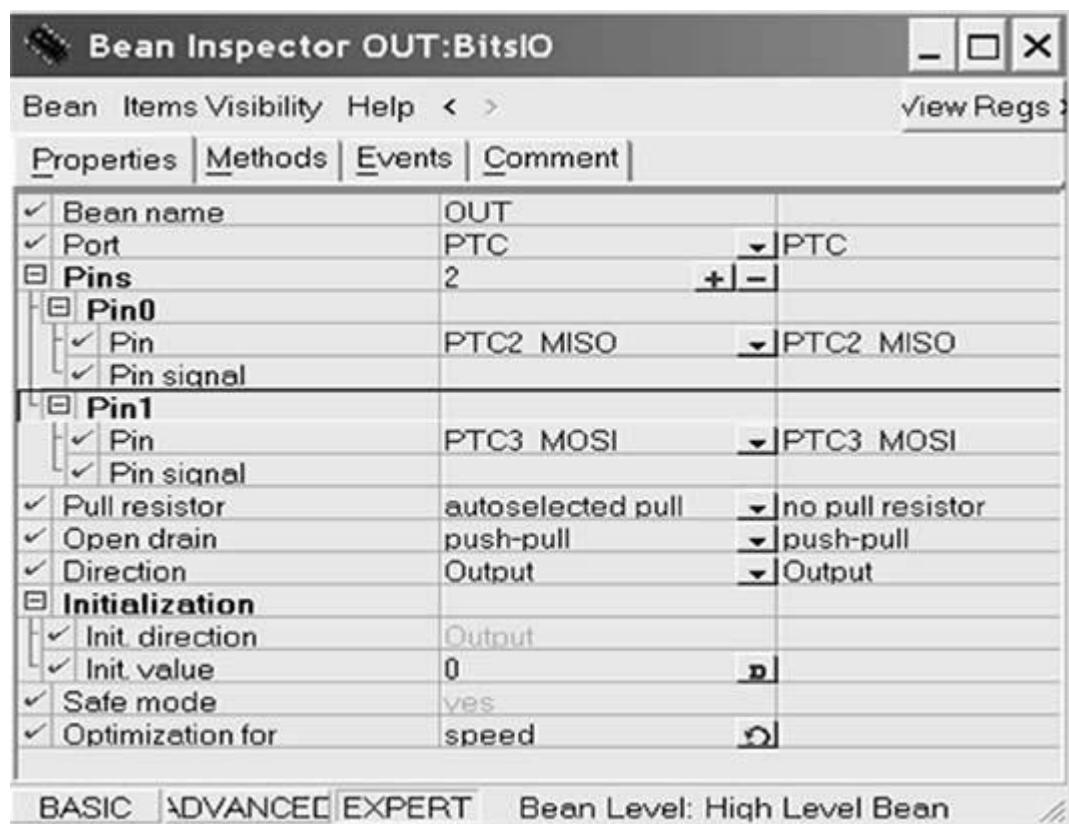
Configuración del “bean” INPUT:

Se adiciona un nuevo “bean” en el módulo **KBI**, que se llamará **INPUT** para el manejo de las entradas **INPUT_1** en **PTD1** y de **INPUT_2** en **PTD2**, los métodos **GetVal()**, **SetEdge()** y eventos **OnInterrupt()** sugeridos pueden ser incluidos en el proyecto, quedando así:



Configuración del “bean” OUTPUT:

Se adiciona un nuevo “bean” que por facilidad se llamará **OUT**, que incorpora los bits **PTC 2** y **PTC 3** que corresponden a las salidas **OUT_1** y **OUT_2**.



En el folder de Métodos, solo habilitar los requeridos: Encender, Apagar y Negar así:



Bean Inspector OUT:BitsIO

Bean Items Visibility Help < >

Properties Methods | Events | Comment |

<input checked="" type="checkbox"/> GetDir	don't generate code	
<input checked="" type="checkbox"/> SetDir	don't generate code	
<input checked="" type="checkbox"/> SetInput	don't generate code	
<input checked="" type="checkbox"/> SetOutput	don't generate code	
<input checked="" type="checkbox"/> GetVal	don't generate code	
<input checked="" type="checkbox"/> PutVal	don't generate code	
<input checked="" type="checkbox"/> GetBit	don't generate code	
<input checked="" type="checkbox"/> PutBit	don't generate code	
<input checked="" type="checkbox"/> SetBit	generate code	
<input checked="" type="checkbox"/> ClrBit	generate code	
<input checked="" type="checkbox"/> NegBit	generate code	

Configuración del “bean” serial al PC:

Crear un nuevo “bean” de serial asincrónico, con las siguientes propiedades:

Bean Inspector SerialPC:AsynchroSerial

Bean Items Visibility Help < > View Regs

Properties | Methods | Events | Comment |

✓ Bean name	SerialPC
✓ Channel	IRSCI
✗ Interrupt service/event	Enabled
✓ Interrupt RxD	INT IRSCIRxReceive
✓ Interrupt RxD priority	medium priority
✓ Interrupt TxD	INT IRSCITxTransmit
✓ Interrupt TxD priority	medium priority
✓ Interrupt Error	INT IRSCIError
✓ Interrupt Error priority	medium priority
✓ Input buffer size	20
✓ Output buffer size	30
✗ Handshake	
✗ CTS	Disabled
✗ RTS	Disabled
✗ Settings	
✓ Parity	none
✓ Width	8 bits
✓ Stop bit	1
✗ Receiver	Enabled
✓ RxD	PTC7 SCRxD
✓ RxD pin signal	
✗ Transmitter	Enabled
✓ TxD	PTC6 SCTxD
✓ TxD pin signal	
✓ Baud rate	9600 baud
✓ Break signal	Disabled
✓ Wakeup condition	Idle line wakeup
✓ Transmitter output	Not inverted
✓ Idle line mode	starts after start bit
✗ Infrared SCI output	disabled
✗ Initialization	
✓ Enabled in init. code	yes
✓ Events enabled in init.	yes
✗ CPU clock/speed sele	
✓ High speed mode	This bean enabled
✓ Low speed mode	This bean disabled
✓ Slow speed mode	This bean disabled

BASIC

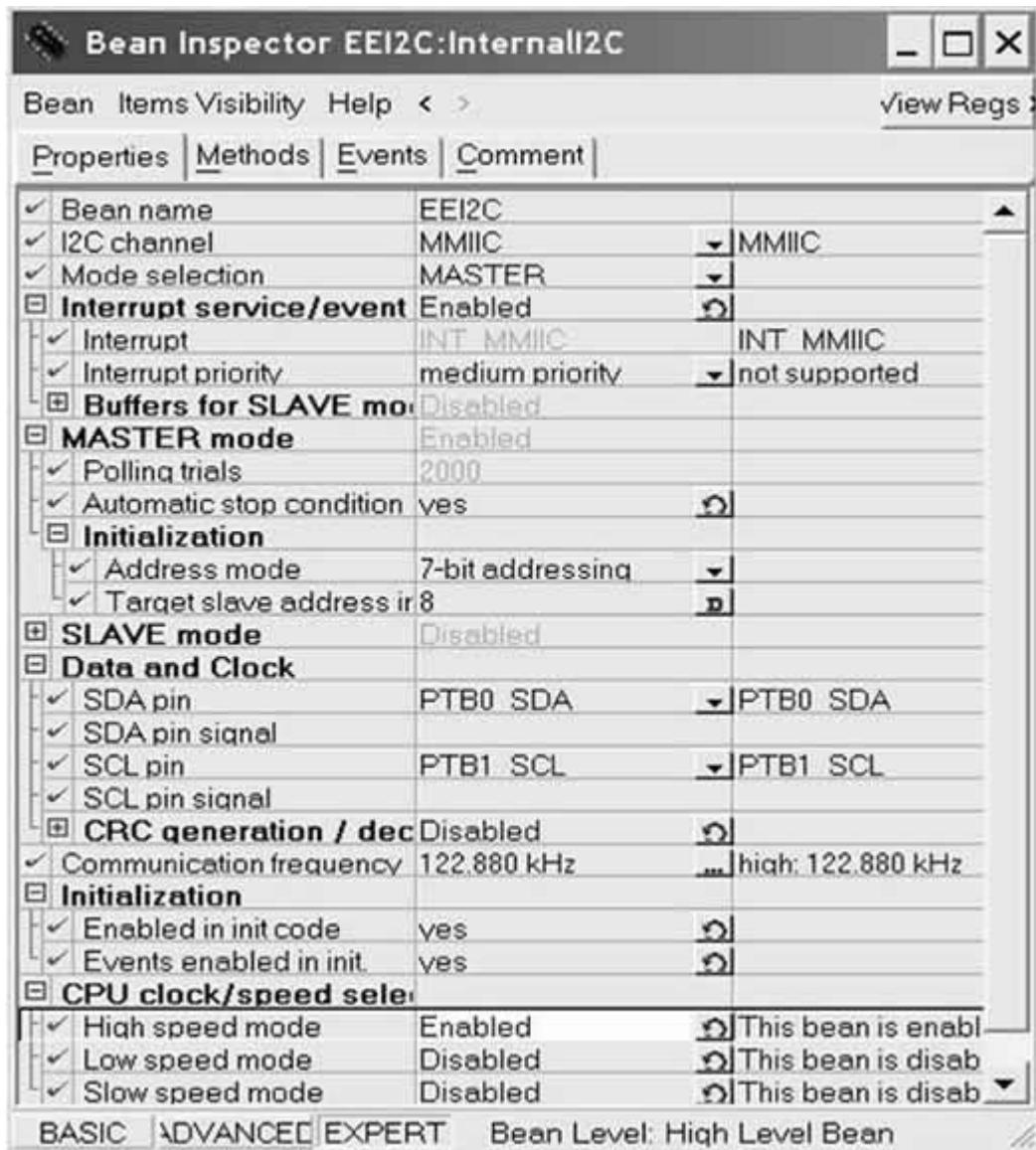
ADVANCED

EXPERT

Bean Level: High Level Bean

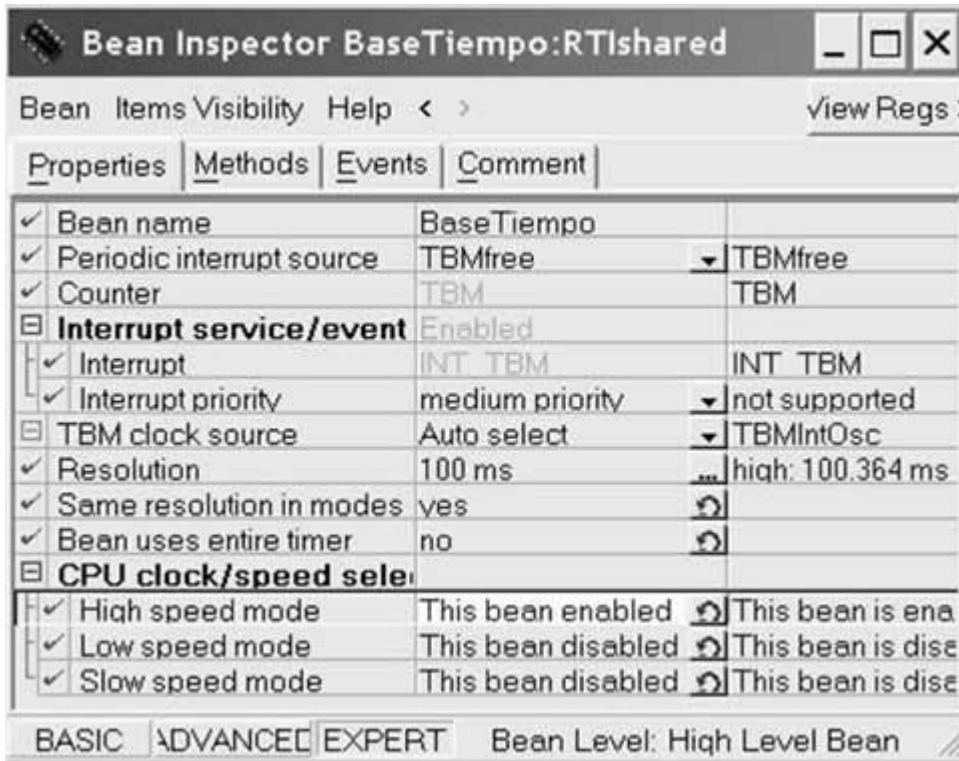
Configuración del “bean” I2C EEPROM M:

Se crea un nuevo “bean” llamado EEI2C, para el manejo de la memoria externa E2Prom:



Configuración del “bean” BaseTiempo:

Se crea un nuevo “bean” para el manejo de una Base de Tiempo de 100mSeg.



Y se genera el código en **Processor Expert → Generate Code ‘EPRO M.mcp’**.

El código del módulo principal que resuelve el enunciado se verá de la siguiente forma:

```
/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "ValorExt.h"
#include "SerialPC.h"
#include "EEI2C.h"
#include "BaseTiempo.h"
#include "Out_1.h"
#include "Out_2.h"
#include "Input_1.h"
#include "Input_2.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include <stdio.h>
#include <string.h>
char bufferMem[5],buffer[30],cntBlink,endTxEEI2C,endRxEEI2C,serialPCTxOk;
unsigned char datoI2C, countSample;
unsigned int datoRx;
word nroBytes,valorAdc;
#define MEM24XX04_SLAVE 0x50 /*dirección memoria E2Prom*/
```

```

#define VALOR_ADC_2V ((2*65535)/5) /*Conversión para 2 voltios*/
#define VALOR_ADC_4V ((4*65535)/5) /*Conversión para 4 voltios*/
void ValorExt_Test(void){ /*Función de comparación de valor ADC*/
    (void)ValorExt_Measure(TRUE);
    (void)ValorExt_GetValue16(&valorAdc);
    if(valorAdc < VALOR_ADC_2V){
        if(cntBlink){ cntBlink = 0; Out_1_NegVal(); }
    }else{ Out_1_ClrVal(); }
}
void BufferPC_Init(void){ /*Función de borrado del buffer de tx de datos*/
unsigned char i;
for(i=0;i<30;i++) buffer[i] = 0;
}
void main(void){ //Programa Principal
/* Write your local variable definition here */
/** Processor Expert internal initialization.
    DON'T REMOVE THIS CODE!! ***/
PE_low_level_init();
/** End of Processor Expert internal initialization*/
/* Write your code here */
for(;;{
    serialPCTxOk = 0;
    (void)SerialPC_SendBlock("/**Alfaomega Grupo Editor**\n\r",28, &nroBytes);
    while(!serialPCTxOk){ }
    serialPCTxOk = 0;
    (void)SerialPC_SendBlock("Input-1 para tomar muestras\n\r",29, &nroBytes);
    while(!serialPCTxOk){ }
    /*Toma de valores leidos del ADC al presionar Input-1*/
    for(countSample = 1,bufferMem[0] = 0x00;countSample <= 20; countSample++){
        do{ ValorExt_Test(); }while(Input_1_GetVal());
        (void)ValorExt_Measure(TRUE);
        (void)ValorExt_GetValue16(&valorAdc);
        bufferMem[1] = *((char *)&valorAdc);
        bufferMem[2] = *((char *)&valorAdc)+1;
        (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
        (void)EEI2C_SendBlock((void *)bufferMem,3,&nroBytes);
        bufferMem[0] += 2;
        BufferPC_Init();
        (void)sprintf(buffer,"r\nMuestra[%u]=%u\r\n",countSample,valorAdc);
        serialPCTxOk = 0;
        (void)SerialPC_SendBlock(buffer,30,&nroBytes);
        while(!serialPCTxOk){ }
        do{ ValorExt_Test(); }while(!Input_1_GetVal());
    }
    //Fin toma de muestras, espera presión de Input-2
    serialPCTxOk = 0;
    (void)SerialPC_SendBlock("Input-2 envio de muestras\n\r",27, &nroBytes);
    while(!serialPCTxOk){ }
}

```

```

do{ValorExt_Test();}while(Input_2_GetVal());
do{ValorExt_Test();}while(!Input_2_GetVal());
//Envio de datos tomados vía serial
for(countSample = 1,bufferMem[0] = 0x00;countSample <= 20; countSample++){
    (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
    (void)EEI2C_SendChar(bufferMem[0]);
    do{ValorExt_Test();}while(Input_1_GetVal());
    (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
    (void)EEI2C_RecvBlock((void*)&datoRx,2,&nroBytes);
    do{ ValorExt_Test(); }while(!Input_1_GetVal());
    BufferPC_Init();
    (void)sprintf(buffer,"r\nData ADC[%u]= %u\r\n",countSample,datoRx);
    serialPCTxOk = 0;
    (void)SerialPC_SendBlock(buffer,30,&nroBytes);
    while(!serialPCTxOk){}
    bufferMem[0] +=2;
}
/**Don't write any code pass this line, or it will be deleted during code
generation.*/
/** Processor Expert end of main routine. DON'T MODIFY THIS CODE!! ***/
for(;;){}
/** Processor Expert end of main routine. DON'T WRITE CODE BELOW!! ***/
} /** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/
/* END EEPROM */
/* This file was created by UNIS Processor Expert 3.03 [04.07]
   for the Freescale™ HC08 series of microcontrollers. */

```

En el modulo **Events.C** se modifican las siguientes funciones, con el objetivo de notificar al programa principal los sucesos de fin de trasmisión serial,recepción y trasmisión a la memoria E2Prom así:

```

extern char endTxEEI2C,endRxEEI2C;
void EEI2C_OnTransmitData(void){ /* Write your code here ... */
    endTxEEI2C = 1;
}
extern char serialPCTxOk;
void SerialPC_OnFreeTxBuf(void){ /* Write your code here ... */
    SerialPC_ClearTxBuf();
    serialPCTxOk = 1;
}
void EEI2C_OnReceiveData(void){ /* Write your code here ... */
    endRxEEI2C = 1;
}

```

Y el modulo **BaseTiempo.C** requiere la modificación de la función ISR, para que indique mediante la línea cntBlink = 1, el suceso de la interrupción que notifica al programa principal sobre este suceso,que permite el encendido y apagado de las

```

salidas OUT-1 y OUT-2 de forma periódica
extern char cntBlink;
ISR(BaseTiempo_Interrupt){
    Cnt++; /* Increment SW counter Cnt */
    if (Cnt == 0x0F) { /* Is it now the period time? */
        Cnt = 0; /* Reset SW counter Cnt */
        cntBlink = 1;
    }
    TBCR_TACK = 1; /* Reset interrupt request flag */
}

```

En el modulo **Cpu.C** asegurarse que el registro MOR este inicializado como se indica
/ Initialization of the CPU registers in FLASH */*

```

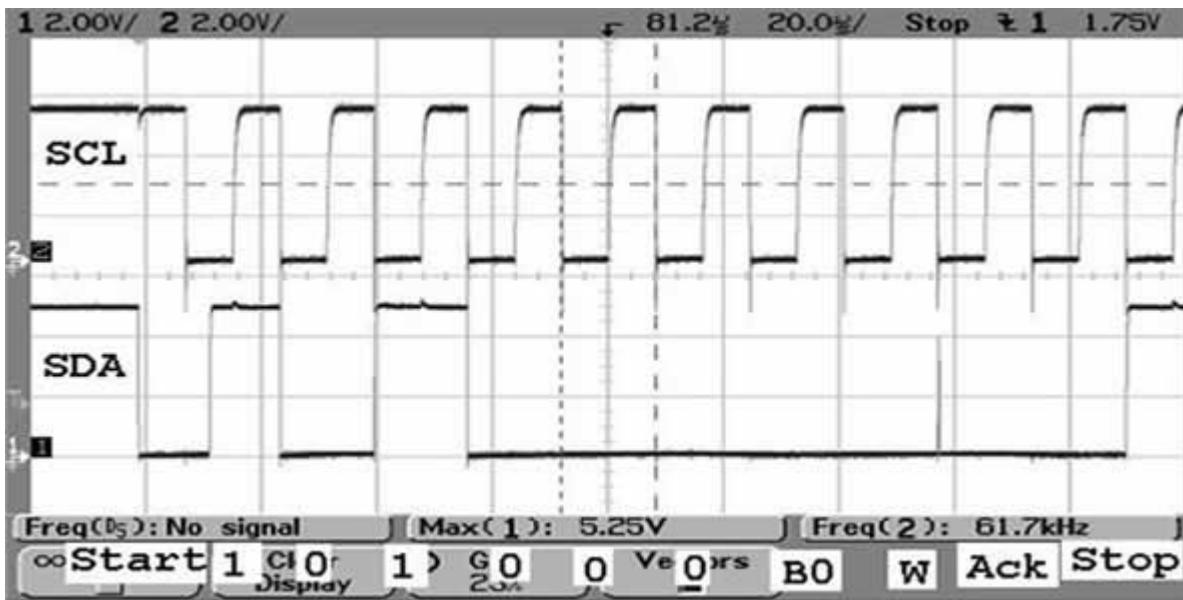
/* MOR: OSCSEL1=1,OSCSEL0=1,??=1,??=1,??=1,??=1,??=1,??=1 */
const unsigned char MOR_INIT @0x0000FFCF = 0xFF;

```

Esta línea configura el oscilador externo como fuente de reloj, puede ocasionar que el programa funcione muy bien mientras se realice depuración, pero no cuando se ejecuta la aplicación en modo RUN .

Discusión:

Inicialmente se identifican los módulos internos y externos que deben crearse haciendo uso del “Processor Expert™”, se configuran uno a uno con base en las especificaciones del enunciado, velocidades de transmisión, dirección de pines y métodos y eventos requeridos. El módulo de manejo de la memoria serial EEI2C, es ensayado de forma independiente, escribiendo un mensaje conocido en ella, en este caso el mensaje “Lenguaje C” y se realiza de nuevo su lectura, para verificar que las funciones generadas realizan lo esperado. La señal de inicio de trama como lo indica la especificación I2C y la hoja de datos de la memoria 24LC04 de Microchip™, indica que debe primero enviarse un señal de START, que consiste en una transición de la línea SDA de alto a bajo mientras la señal de SCL está en alto, y seguir la secuencia 1 0 1 0 0 0 B0 donde B0 es el banco de memoria, que en este caso es cero, de parte de la memoria serial se debe recibir una señal de reconocimiento (ACK) en cero, como se muestra en el siguiente diagrama de tiempo.



El mensaje enviado se realiza con la función **EEI2C_SendBlock()** con los argumentos apropiados.

El código de prueba para la memoria serial luce de la siguiente forma:

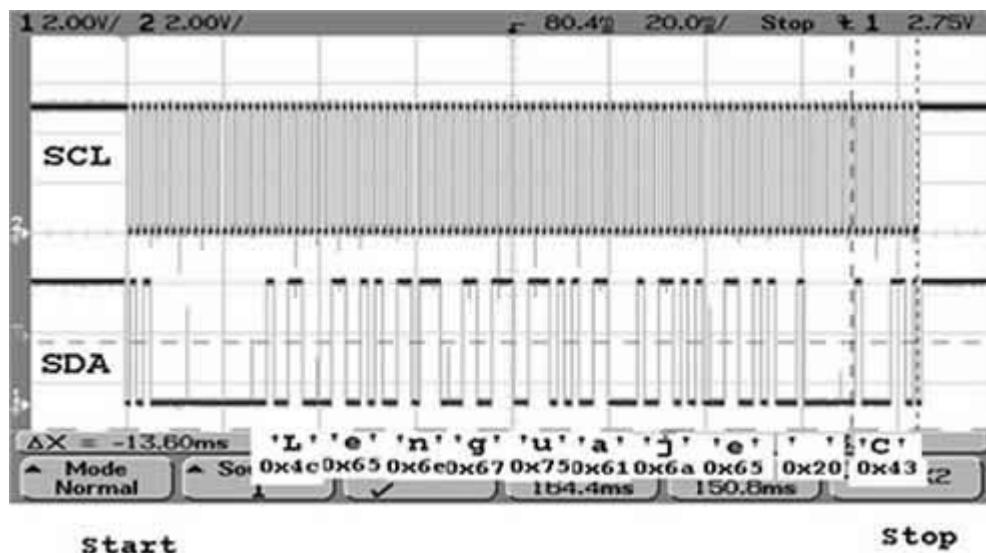
```

const char bufferMem[] = { 0x00, //dirección 'L','e','n','g','u','a','j','e',' ', 'C' };
unsigned char datoI2C;
char datosRx[20];
word nroBytes;
#define MEM24XX04_SLAVE 0x50

void main(void){
    /* Write your local variable definition here */
    /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!!
***/
    PE_low_level_init();
    /*** End of Processor Expert internal initialization. ***/
    /* Write your code here */
    (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
    (void)EEI2C_SendBlock((void *)bufferMem,11,&nroBytes);
    while(Input_1_GetVal()){ }
    (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
    (void)EEI2C_SendChar(0x00);
    while(Input_2_GetVal()){ }
    (void)EEI2C_SelectSlave(MEM24XX04_SLAVE);
    (void)EEI2C_RecvBlock((void*)datosRx,5,&nroBytes);
    for(;;){
        if(!Input_1_GetVal()){ nroBytes = datoI2C; }
    }
}

```

El ciclo de escritura muestra el siguiente diagrama de tiempos entre el SCL y el SDA.



Que corresponde al código:

```
(void)EEI2C_SelectSlave(MEM24XX04_SLAVE);  
(void)EEI2C_SendBlock((void *)bufferMem,11,&nroBytes);
```

Materiales
adicionales en la



Escrutura y lectura memoria
E2PROM: simulación y proyecto
completo para Freescale™
(AP-Link).

Se puede observar que el manejo de hardware
y de los módulos internos del

microcontrolador se realiza de forma muy
sencilla y sin un conocimiento muy detallado,
resolviendo el enunciado en el loop principal
con algunos cambios específicos al código
generado por el “Processor Expert™”.

RESUMEN DEL CAPITULO

El “Processor Expert™” para Freescale™ provee una herramienta con ventajas significativas como:

Facilidad de programación de los periféricos de un microcontrolador, con un conocimiento básico por parte del programador.

Interfaz GUI que configura los módulos en términos del mundo real. Contiene manejadores de hardware (drives), listos para usar los periféricos internos de procesador.

Provee una solución de software con funcionalidad en tiempo real.

Permite crear librerías complejas que luego pueden ser compartidas.

Permite que los programadores se concentren en capas superiores de programación en lugar de invertir tiempo en entender la máquina.

Los costos de desarrollo de proyectos embebidos están ligados al costo del desarrollo del software y de la curva de aprendizaje de un procesador moderno y de su software. Con la herramienta “Processor Expert™” que provee el Codewarrior®, se puede realizar prototipos y proyectos con facilidad y con un mínimo conocimiento de un procesador, dando libertad de poder portar un proyecto a un procesador de 8,16, 32 bits o DSP, manteniendo el código de aplicación intacto.

PREGUNTAS Y EJERCICIOS PROPUESTOS

Cuáles son los beneficios más importantes del usar el “Processor Expert™”?

¿En qué circunstancias no sería recomendable usar el “Processor Expert™” para un proyecto de sistema embebido?

¿Cómo se puede maximizar el uso del “Processor Expert™” sin que afecte el espacio de memoria utilizado en el microcontrolador?

Usando “Processor Expert™” y el sistema AP_Link desarrollar una aplicación que imprima en un LCD 20x2, los datos que llegan por el serial SCI1, y en memoria EEPROM externa, el número de caracteres que se han recibido (usar un **unsigned long** para su almacenamiento).

Usando “Processor Expert™” y el sistema AP_Link desarrollar una aplicación que recibe datos por el puerto serial SCI1 y los transmite por el SCI2 en ascii mayúsculas y viceversa, lo que se recibe por SCI2 se trasmite por SCI1 en minúsculas.

Crear un proyecto con Processor Expert™, que utilice el **bean** creado en el Ejemplo 33. Ensayar con varios valores para que la generación del código sea la correcta.

INTRODUCCIÓN

El desarrollo de software de un proyecto embebido puede tener muchas aproximaciones, y dependiendo de la complejidad del proyecto es necesario involucrar técnicas que faciliten la migración entre diferentes tecnologías, marcas y plataformas, la reutilización del código y el trabajo en equipo.

Gracias al nivel de integración actual, la disminución de precios y las nuevas tecnologías que permiten tener mayores velocidades de ejecución en los procesadores embebidos, el uso de un sistema operativo de tiempo real es adoptado cada vez más por los grupos de programadores alrededor del mundo.

El uso de un RTOS (*Real Time Operating System*) en el diseño de un sistema embebido tiene gran cantidad de ventajas sobre la forma tradicional de programación, especialmente cuando el grupo de R&D (Research and Development) lo conforman varios diseñadores, porque facilita el mantenimiento del software e independiza a los programadores de la coordinación de varias tareas y del manejo de funciones que dependen del tiempo.

Los sistemas operativos ofrecen un entorno estándar para que el software pueda interactuar con el hardware, suelen ser uniformes, esto les permite funcionar como plataformas en un entorno en el cual las aplicaciones son desarrolladas, de este modo se asegura que una aplicación ejecutada en un ordenador se ejecutará en otro, incluso si éste tiene diferentes recursos.

En este capítulo se muestran dos técnicas que han sido adoptadas por varios programadores: la primera, conocida como loop Consecutivo/Interrupción (*Foreground/Background*), la segunda usa uno de los sistemas operativos más populares llamado el µC/OS-II de la compañía micrium. La primera técnica, aunque no es un sistema operativo como tal, resulta bastante eficiente para procesadores de 8 bits, y se puede complementar con el uso de varios conceptos de RTOS, mientras que la segunda incorpora al proyecto un código muy robusto con posibilidades de expansión y garantía del tiempo real.

Este tema se trata de forma práctica con dos ejemplos reales que muestran su funcionalidad y forma de trabajo dentro del ambiente microcontrolado.

10.1 ¿QUÉ ES UN SISTEMA OPERATIVO DE TIEMPO REAL?

Los sistemas operativos ofrecen un entorno estándar para que el software pueda interactuar con el hardware, suelen ser uniformes, esto les permite funcionar como plataformas en un entorno en el cual las aplicaciones son desarrolladas, de este modo se asegura que una aplicación ejecutada en un ordenador se ejecutará en otro, incluso si éste tiene diferentes recursos.

El RTOS es a su vez un programa que ocupa recursos del procesador, como son espacio en memoria de programa Flash, memoria RAM, base de tiempo (*timer*), y además requiere tiempo para tomar decisiones y manejar las diferentes tareas. Por esta razón, a la familia de procesadores de 8 bits puede afectarles el hecho de incorporar un sistema de estos, y a ellos se dedicará mayor atención en esta sección; una vez el sistema funcione de forma aceptable en 8 bits, su desempeño mejorará en procesadores



Es bueno programar como si la máquina receptor fuese de 8 bits, porque para estos procesadores el programador ha de tener en cuenta el tamaño y la velocidad de ejecución que pueden ser críticas;

superiores como son los de 16 y 32 bits.

En síntesis un sistema operativo de tiempo real RTOS (*Real Time Operating System*), es un programa que coordina el funcionamiento de otros programas llamados tareas. El sistema operativo permite, de forma óptima, el manejo de datos, ejecución de funciones, generación y control de eventos y coordinación general de un sistema.

si luego se pasa a un procesador de 16 o 32 bits el software correrá con mejor desempeño.



El sistema operativo es principalmente un conjunto de programas que se ejecutan y gestionan el funcionamiento de todos los componentes de la computadora, como el monitor o pantalla, la unidad central de procesamiento, los periféricos de la computadora, el hardware, los controladores y todas las aplicaciones instaladas en el ordenador. Un asistente personal digital, un Smartphone, un teléfono celular, un reproductor de MP3 y otros dispositivos de mano tienen sus propios sistemas operativos y también son la base de computadoras personales o portátiles.

El RTOS tiene la capacidad para adaptarse a las diferentes necesidades y exigencias, puede reconfigurarse o cambiarse totalmente si las tareas son más complejas y adaptarse para llevar a cabo una tarea diferente o cuando aparezcan nuevos programas o hardware que va a ejecutarse en la máquina.



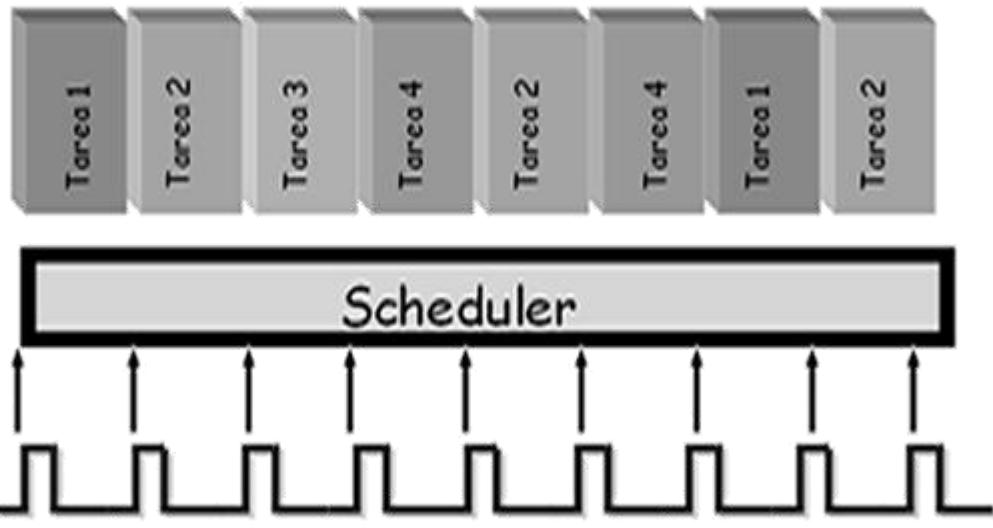
El sistema operativo es un conjunto de programas que ejecutan y gestionan el funcionamiento de todos los componentes de la computadora o máquina.



El RTOS ocupa recursos del procesador para la coordinación de las tareas tanto en tiempo de ejecución como en memoria. El método más común de coordinar la ejecución de las diferentes tareas consiste en incluir en el RTOS un componente llamado el **scheduler**, el cual se encarga de entregar pequeños lapsos de tiempo a cada una de las tareas, suspender su ejecución y pasar el control a la siguiente tarea en lista, de esta forma todas las tareas van ejecutando sus labores.

El tiempo y el cambio de tarea (*context switching*) son gobernados por un reloj externo al RTOS llamado el **TICK**, que consiste en una base de tiempo generada a partir de una interrupción periódica del microcontrolador (*ver Gráfico 10.1*).



GRÁFICO
10.1

De esta forma cada tarea que es suspendida por pequeños lapsos de tiempo, percibe que el procesador esta ejecutándola continuamente, cuando en realidad el procesador está suspendiendo la ejecución de la tarea y regresa.

Los sistemas operativos modernos permiten a la unidad central de procesamiento realizar procesos de multitarea o ejecutar múltiples procesos. El sistema operativo no puede ejecutar al mismo tiempo los procesos, sino que cambia rápidamente de un proceso a otro, ejecutándolos por lo general de acuerdo con las prioridades de funcionamiento del sistema de gestión de los procesos de algoritmo.

10.2 Terminología básica sobre RTOS

10.2.1 Tareas (*Task ó Thread*)



Las tareas son procedimientos formados por códigos de programa que asumen que la CPU está disponible solo para operarla a ella. Las tareas trabajan con base en estados y eventos.

Una tarea es encargada de realizar una labor específica en la aplicación embebida; para su avance, el sistema operativo le entrega pequeñas porciones de tiempo invocando la tarea de forma consecutiva en el punto en el cual la suspendió la vez anterior, dando la impresión de continuidad.

Una tarea en un sistema embebido puede ser la encargada de la inicialización, control y manejo del sistema de visualización LCD, otra tarea podría ser responsable por el manejo del sistema de teclado, otra la encargada de controlar el módulo de comunicaciones seriales con un dispositivo externo, etc.

Prioridad de ejecución de una tarea

Las tareas tienen asociadas ciertas características o atributos, que pueden ser definidas al momento de su creación, o cambiar de forma dinámica a medida que ella se ejecuta, suceden interrupciones o se ejecutan otras tareas.

La prioridad de ejecución se refiere a la importancia que una tarea tiene sobre la ejecución de las demás; esta importancia puede definirse en un sistema con 3 niveles: **BAJA, MEDIA y ALTA**, o bien con un número (ejemplo: entre 0 y 62), que define la prioridad de la tarea, el valor más bajo en número de prioridad, indica mayor importancia; así, una tarea que tenga prioridad 3, tiene mayor importancia para ejecutarse que la que tiene prioridad 10.

La prioridad puede ser fija, definida al principio de la ejecución del sistema, o bien manejarse de forma dinámica en el transcurso del programa.

Dependiendo de la naturaleza del RTOS usado, varias tareas pueden tener en un mismo momento la misma prioridad y en este caso el *scheduler* se encarga de hacer llamado secuencial a cada una de ellas en un esquema denominado **Round-Robin**; sin embargo, algunos sistemas solo permiten una prioridad única por tarea y la que se ejecuta en determinado momento es la que tiene la mayor prioridad, mientras que las demás deben esperar que la tarea de mayor prioridad ejecute su código.



El método más común de coordinar la ejecución de las diferentes tareas consiste en incluir en el RTOS un componente llamado el scheduler, el cual se encarga de entregar pequeños lapsos de tiempo a cada una de las tareas para crear el efecto de simultaneidad.

El sistema operativo, no puede ejecutar al mismo tiempo los procesos, sino que cambia rápidamente de un proceso a otro, ejecutándolos de acuerdo con las prioridades de funcionamiento diseñadas por el programador.



La ejecución se

refiere a la importancia que una tarea tiene sobre la ejecución de las demás; esta importancia puede definirse en un sistema con 3 niveles: BAJA, MEDIA y ALTA, o bien con un número (ejemplo: entre 0 y 62).

Una tarea puede variar su prioridad a medida que corre el programa; por ejemplo, el manejo del display puede ser una tarea de prioridad BAJA si el teléfono está inactivo, pero puede pasar a ser de prioridad ALTA cuando se tiene una llamada activa.

Pero la prioridad puede ser cambiada de forma dinámica, y esto sí lo permite la gran mayoría de los RTOS comerciales. En un sistema multitarea que administra un equipo celular, puede ser de prioridad ALTA el manejo de la batería cuando su nivel está por debajo de un nivel crítico. Sin embargo, puede pasar a ser una tarea de prioridad MEDIA si el nivel está en un rango seguro y tal vez pasar a prioridad BAJA si el equipo está conectado a la red eléctrica.

El manejo del display puede ser una tarea de prioridad BAJA si el teléfono está inactivo, pero puede pasar a ser de prioridad ALTA cuando se tiene una llamada activa, realiza la reproducción de un archivo de video o una teleconferencia.

Estados de tarea

Una tarea puede encontrarse en diferentes situaciones, dependiendo de su estado en ejecución (*Gráfico 10.2*):

Suspendida (Dormant): indica que una tarea está creada o terminada y no está siendo ejecutada por el RTOS ni programada su ejecución por el *scheduler*.

Ejecución (Run): una tarea se encuentra en este estado cuando el scheduler del RTOS ha pasado el control a su código y está siendo ejecutada en el presente por la CPU.

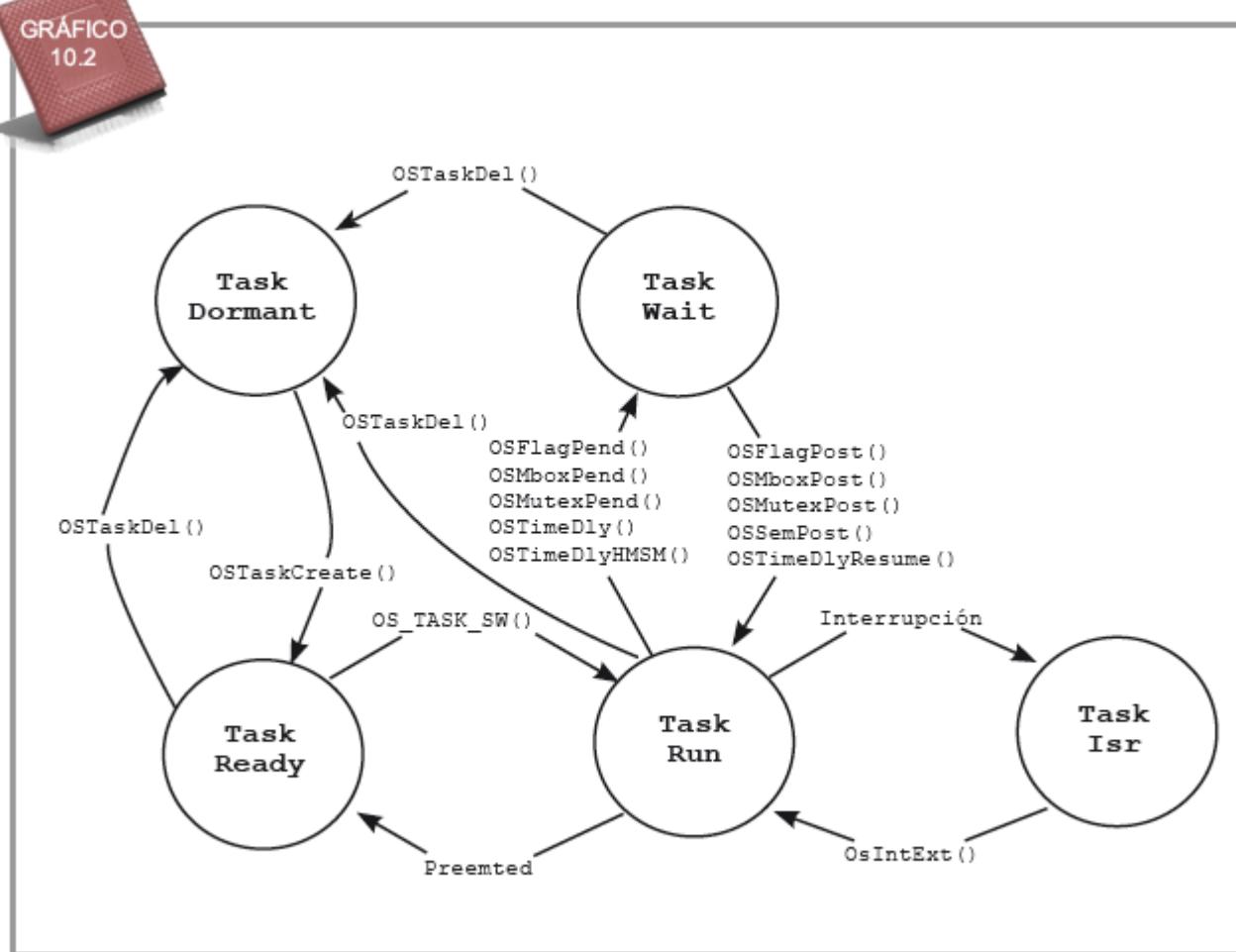
Disponible (Ready): una tarea que está esperando turno para ser ejecutada por el scheduler. Su prioridad es más baja que la que está en modo **Run**.

Espera (Wait): estado en el que una tarea está a la espera de algún evento para continuar su ejecución, hasta que no se presente dicho evento no estará en la lista de tareas que son ejecutadas por el *scheduler*. El evento que espera la tarea puede ser el cambio de un pin externo, que un recurso esté disponible, que pase un determinado tiempo, que algún proceso de otra tarea termine, etc.

Interrumpida (Interrupted): cuando una tarea estando en modo Run, es suspendida por la aparición de una **ISR**. En este caso el **RTOS** no tiene participación en la administración de suspender y retornar el control a la tarea.



Estado básico de las tareas en un RTOS¹.



1 Gráfico cortesía www.micrium.com



10.2.2 Recursos (*resources*)



Los recursos
compartidos
con exclusión
mutua no permiten

ser accesados por dos tareas de forma simultánea; ejemplo de ello es un disco duro para almacenamiento de información, un display de visualización de texto o gráficos o una impresora externa.



Los recursos son entidades usadas por una tarea. Un recurso puede ser una función no reentrant, una estructura, una variable, un periférico interno del microcontrolador, un dispositivo externo como una memoria serial, un display, un teclado, una impresora, un sistema de almacenamiento, etc. Se dice que un recurso es compartido (*shared resource*), cuando es usado por más de una tarea, debido a que cada tarea puede tener acceso exclusivo al recurso en un determinado espacio de tiempo, y prevenir que otra tarea lo haga, con el objetivo de prevenir daños en los datos o en los dispositivos, este proceso es denominado exclusión mutua (*mutual exclusión*). Casos típicos de recursos compartidos con exclusión mutua son un disco duro para almacenamiento de información, un display de visualización de texto o gráficos, una impresora externa, entre otros.



10.2.3 Eventos (*events*)

Son elementos de notificación hacia una tarea para indicar la ocurrencia de un hecho o estímulo que interesa a una o varias tareas que se están ejecutando. Los eventos pueden llegar ya sea del hardware interno o externo al procesador, o bien pueden ser eventos de software: *timeouts*, interrupciones o un resultado de la operación de una función.



10.2.4 Semáforos (*semaphores*)

Los semáforos son mecanismos de control que proveen al RTOS una forma de prevenir que múltiples tareas realicen acceso al mismo recurso en el mismo tiempo. Se usan además para indicar la ocurrencia de un evento, y permiten sincronizar varias actividades dentro del sistema. Una vez una tarea va a iniciar un proceso con un recurso, pregunta por el estado del semáforo; si el semáforo está activo la tarea quedará en el estado de *wait*, si no está activo, lo toma y lo bloquea, realiza su operación en el recurso y una vez termina, libera el semáforo, permitiendo que otras tareas a partir de ese momento puedan hacer uso del recurso de la misma forma.



10.2.5 Mensajes (*message mailbox*)

Un mensaje es un objeto que se entrega a una tarea, o bien puede ser una ISR², un apuntador con determinada medida a otra tarea. Este apuntador normalmente esta inicializado a una estructura de datos que contiene un mensaje. Existen en el sistema operativo funciones ya incorporadas que permiten crear, dejar el mensaje pendiente, aceptar, preguntar si existe un mensaje específico para alguna tarea.

2 ISR: Interrupt Service Routine.



10.2.6 Bloques de Memoria (*buffers*)

Son elementos de almacenamiento continuos para la capa de aplicación, permite crear listas circulares, FIFO³ o LIFO⁴ de cualquier tamaño. Las listas pueden crearse y borrarse de forma dinámica, permitiendo de esta forma un manejo eficiente de la memoria. Funcionan de forma similar a las funciones *malloc()* y *free()* para el manejo dinámico de memoria.

-
- 3 FIFO: First Input First Output.
 - 4 LIFO: Last Input First Output.



10.2.7 Reloj y Timers (*Clock Tick & Timers*)

El “clock tick” o tick es una interrupción que ocurre de forma periódica, le permite al RTOS contar el tiempo que entrega a cada tarea y el manejo de timeouts.

El valor del tick de frecuencia más alta genera mayor preámbulo (*overhead*) en el sistema y tiene un mayor consumo de energía, debido a que debe tomar decisiones más veces por segundo, lo cual puede ser crítico para el desempeño de la máquina; sin embargo, un tiempo muy largo de *tick* ocasiona que las tareas tengan tiempo de respuesta más lento y que la sensación de “tiempo real” se vea afectada.

Es responsabilidad del diseñador la selección adecuada del valor del tick dependiendo de los requerimientos de velocidad, consumo y tiempo de respuesta requerido. Valores típicos para un sistema embebido pueden ir desde 10 mseg a 100 mseg, dependiendo de los requerimientos de una aplicación particular.



El valor del tick le permite al diseñador controlar los requerimientos de velocidad, consumo y tiempo de respuesta del sistema.



10.2.8 Kernel

El *kernel* es la parte del RTOS responsable por la distribución del tiempo de la CPU para el manejo de las tareas, además de la comunicación entre ellas. El servicio fundamental del *kernel* es el **cambio de contexto** (véase Capítulo 1.6). El uso de un *kernel* de tiempo real generalmente simplifica el diseño de sistemas permitiendo que la aplicación pueda ser dividida en múltiples tareas que el *kernel* administra una a una.

La parte del *kernel* que se encarga de determinar cual tarea debe ejecutarse es el scheduler, el que basado en la prioridad, pasa el control de la CPU.

Como es sabido, el cambio de contexto agrega preámbulo, debido a que el *kernel* requiere también espacio de tiempo y CPU para realizar el cambio entre tareas, este tiempo debe ser contabilizado y ha de ser insignificante para la aplicación.

Es de esperarse también que para procesadores de 8 bits este porcentaje sea mayor, por las limitaciones de la arquitectura, pero no debería superar más del 5%, en arquitecturas de 16 y 32 bits, este porcentaje no sobrepasa el 2% lo que indica que el sistema operativo será más eficiente en máquinas más robustas.



Cuando se cambia de una tarea a otra, es decir, se produce un cambio de contexto, se agrega un preámbulo, debido a que el kernel requiere también espacio de tiempo y CPU para realizar el cambio entre tareas, y dicho lapso debe ser contabilizado y ha de ser insignificante para la aplicación.



kernel No preemptivo

Un sistema No preemptivo no suspende la tarea que está en estado **Run**, por lo que este tipo de *kernel* es también llamado un sistema cooperativo, porque las tareas entregan el control al sistema de forma voluntaria, sin embargo las interrupciones pueden quitar el control en cualquier momento.

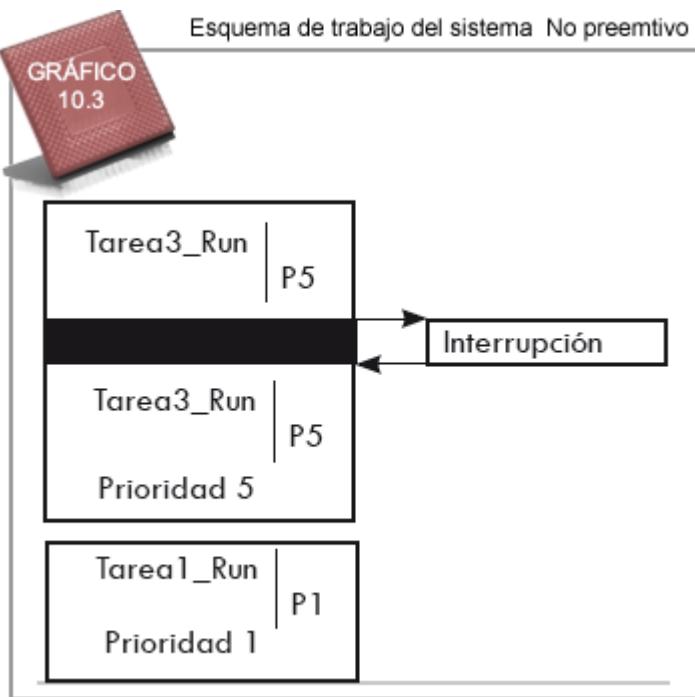
Para mantener la sensación de continuidad el proceso de entregar el control de nuevo a la tarea debe ser frecuente.

En el *Gráfico 10.3* una vez el *scheduler* toma la decisión de pasar el control a la Tarea3 por ser la de mayor prioridad en el momento, ésta inicia su ejecución, en el transcurso sucede una interrupción que le cambia a la Tarea1 su prioridad a 1, la cual es mayor que la que tiene la Tarea3 (prioridad 5); una vez la interrupción retorna, el control se pasa a la Tarea3 (siendo de menor prioridad), y es ésta la que voluntariamente entrega el control al *scheduler*, el cual pasa luego el control a la Tarea 1, que ahora es la de mayor

En un sistema No preemptivo el *kernel* no suspende la ejecución de la tarea actual, sino que las tareas entregan el control al sistema de forma voluntaria; sin embargo, las interrupciones pueden quitar el control en cualquier momento.

prioridad.





Una de las ventajas del sistema No preemptivo es que las interrupciones carecen de preámbulo, debido a que no tienen que tomar decisión alguna sobre a cuál tarea entregarle el control, sin embargo, tiene la desventaja que el tiempo de respuesta de las tareas puede ser un poco más lento, debido a que la tarea que está pendiente de un evento, debe esperar que la tarea de menor prioridad entregue el control y siga su turno, además, el tiempo de respuesta es no determinístico debido a que no se sabe a ciencia cierta cuanto tiempo tardará en entregarse el control. De ello se deduce la importancia que todas las tareas realicen procesos muy rápidos en modo Run, esto evitará que otras tareas más importantes se atrasen.

kernel preemptivo

Un *kernel* preemptivo es aquel que suspende la tarea en estado **Run** y le pasa el control a una de mayor prioridad. El *scheduler* en su labor entrega el control a la tarea de mayor prioridad, esta inicia su ejecución, pero en algún momento de su ejecución, sucede una interrupción que genera el cambio de prioridad o entrega un evento a una tarea de máxima prioridad, en este caso se suspende la tarea a la que se entregó el control y se pasa el control a esta nueva tarea de máxima prioridad.

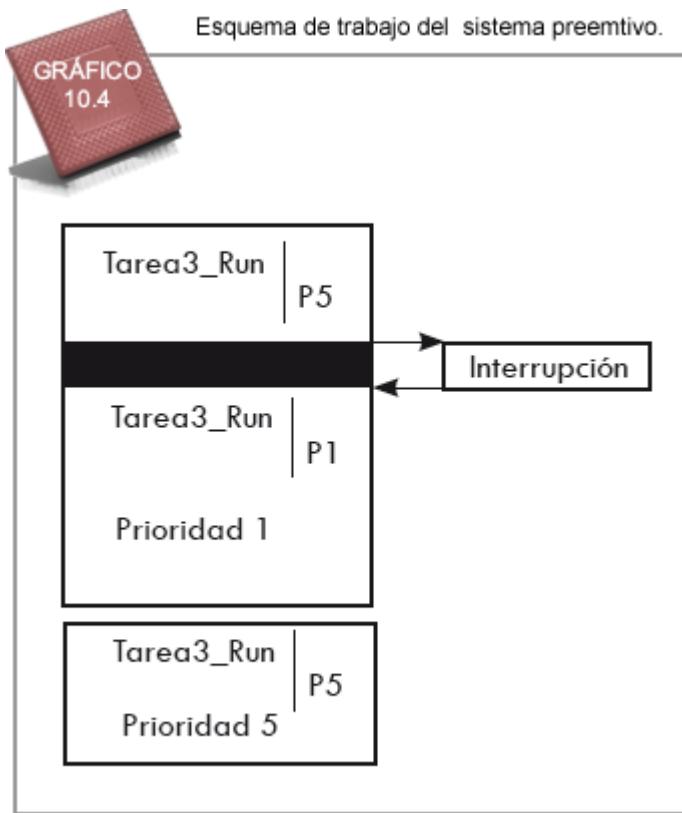


Es importante que todas las tareas realicen procesos muy rápidos en modo Run, esto evitará que otras tareas más importantes se atrasen.

Un kernel preemptivo suspende la tarea en estado Run y le pasa el control a una de mayor prioridad.



Este esquema de trabajo se ilustra en el *Gráfico 10.4*. Inicialmente el *scheduler* para el control a la Tarea3 por ser la de mayor prioridad, mientras está ejecutando esta tarea se presenta una interrupción que cambia la prioridad de la Tarea1 a 1, siendo ésta una prioridad mayor a la de la Tarea3, en este esquema, el control no se pasa a la Tarea3 en estado interrumpida, sino que lo pasa a la Tarea1 por ser ahora de mayor prioridad.



Este esquema tiene la ventaja de garantizar un poco más el “tiempo real” de las tareas de alta prioridad, debido a que la ejecución de la tarea de mayor prioridad es determinístico (se puede medir); además, permite disminuir el tiempo de respuesta de la tareas a un nivel inferior.

Las aplicaciones que usan *kernel* preemptivo tienen más control y requieren mayor manejo de semáforos, debido a que puede darse más frecuente el caso que dos (2) tareas traten de usar el mismo recurso al mismo tiempo.

10.3 SISTEMA DE LOOP CONSECUITIVO/INTERRUPCIÓN

Llamado el sistema *Foreground/Background*, es una técnica bastante usada en sistemas de baja complejidad o de limitación de procesamiento, debido a que no tiene preámbulo (*overhead*) entre cambio de contexto de una tarea a otra.

El cambio se realiza por medio de las instrucciones nativas del procesador, JSR (*jump to subroutine*) o CALL, para entregar el control a una tarea y ponerla en modo **Run** y RTS (*Return From Subroutine*), o RTC (*Return From Call*), para ponerla en estado *Wait* o **Ready**, y continuar con la ejecución de la siguiente tarea en la cola.

10.3.1 Diseño del Loop principal

El diseño del módulo principal consiste en generar un loop infinito en el main (*Gráfico 10.5*) y realizar llamado consecutivo a todas las tareas (*background*) que el sistema requiere procesar, las interrupciones generales se habilitan una vez que los módulo son inicializados, siempre y cuando no se requiera el suceso de interrupciones en los módulo de inicialización permitiendo que interrumpan la ejecución de una tarea; solo se deshabilitan por pequeños lapsos de tiempo en las zonas críticas del programa.

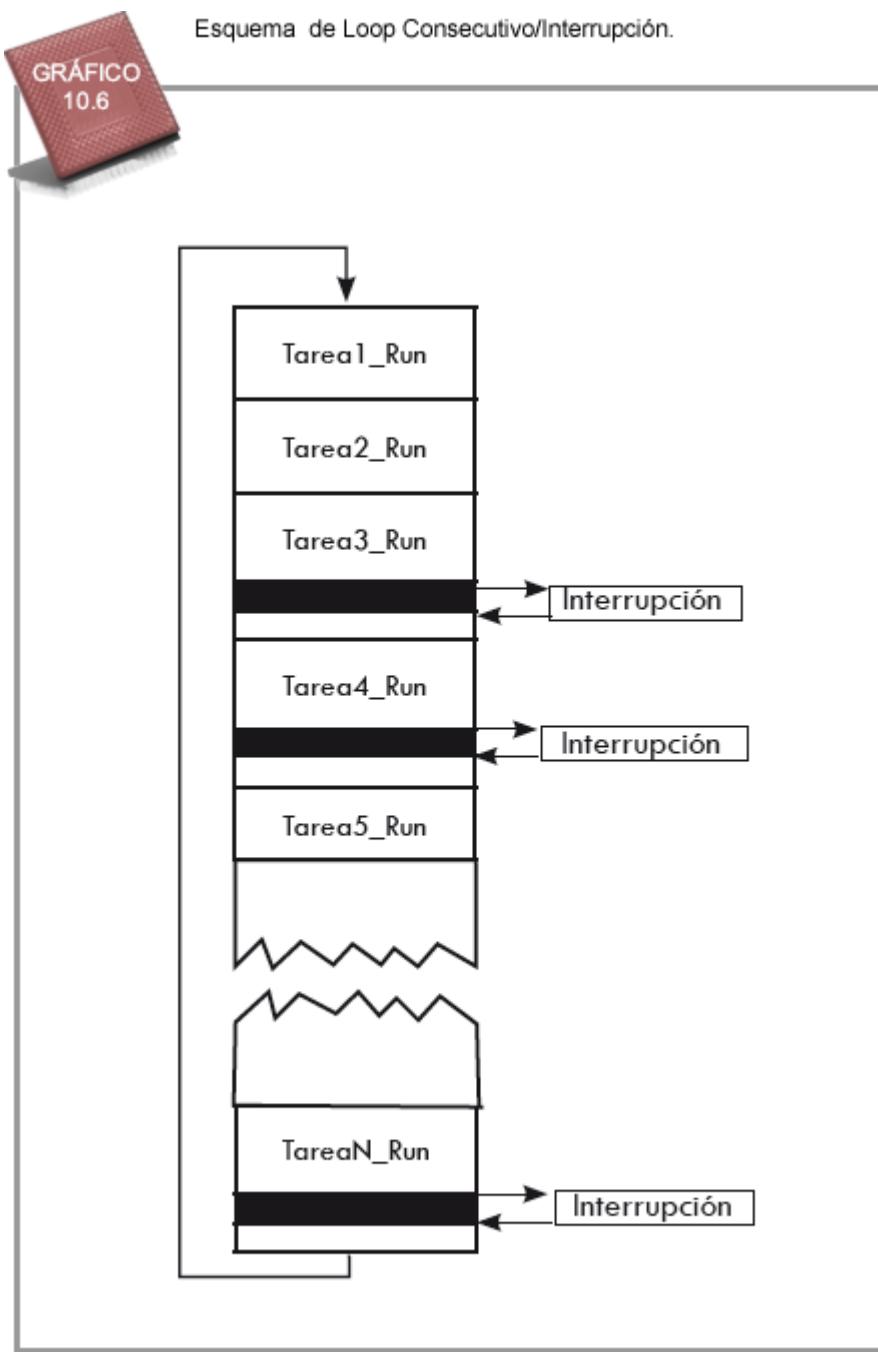


Programa principal esquema Loop Consecutivo/Interrupción.

```
main(void){  
    Mcu_Init();           // inicialización CPU y MCU  
    Tarea1_Init();        // inicialización tareas  
    Tarea2_Init();  
    Tarea3_Init();  
    Tarea4_Init();  
    Tarea5_Init();  
    ...  
    TareaN_Init();  
                           // habilita interrupciones  
    EnableInterrupts;  
  
    for(;;) {             // loop infinito  
        Tarea1_Run();  
        Tarea2_Run();  
        Tarea3_Run();  
        Tarea4_Run();  
        Tarea5_Run();  
        ...  
        TareaN_Run();  
    }  
}  
// fin del módulo principal
```

Las interrupciones se encargan de manejar los eventos asincrónicos (*Foreground*).

El sistema completo consta de 2 niveles de programa: el nivel de tareas (*Task Level*) y el nivel de interrupciones (*Interrupt Level*), como se puede ver en el *Gráfico 10.6*.



En este esquema las operaciones críticas deben ser realizadas por las interrupciones (**ISRs**) para garantizar que los procesos cumplen con los tiempos requeridos. Debido a esto las ISRs tienden a ser un poco más largas de lo que deberían, además que la información para el nivel de tareas no es procesada hasta que llegue su turno de ejecución en el loop consecutivo; esta demora es llamada tiempo de respuesta del nivel de tareas (*Task-Level response*), este tiempo depende de que tanto tiempo tarde el Loop consecutivo en ejecutarse, y puede no ser constante, debido a que depende del procesamiento

que esté llevando a cabo cada tarea independiente, lo que convierte el sistema en un Loop no **determinístico**; inclusive, si el código de una tarea es cambiado por el diseñador, o se agregan nuevas tareas, el tiempo de respuesta a su vez es afectado.

En este esquema, cada tarea está dividida en estados, los cuales se definen al momento del diseño con macros (**#define**) o bien por una lista enumerada (**enum**).

Si se asume que una **TareaX** tendrá **n** estados se crea un par de archivos en el proyecto llamados **TareaX.c** y **TareaX.H**.

10.3.2 Archivo de cabecera TareaX.H

En el archivo **TareaX.H** se agregan los prototipos de las funciones públicas del módulo, definiciones que pueden ser modificadas por el usuario del módulo, tales como pines del microcontrolador asociados al módulo, tiempos de muestreo, timeouts y todo lo que pueda flexibilizar el uso del módulo a futuro.

El archivo de cabecera tiene 3 funciones públicas: el constructor o de inicialización **TareaX_Init()**, la de estado estable o de ejecución **TareaX_Run()** y el destructor **TareaX_Flush()** , el archivo de cabecera tendrá el aspecto que se ilustra en el *Gráfico 10.7*.



El archivo de cabecera tiene 3 funciones públicas: el constructor o de inicialización, la de modo estable o de ejecución y el destructor.





Estructura general del archivo de cabecera TareaX.H.

```
// Archivo de Cabecera del módulo TareaX
//***** TareaX.H *****

#ifndef TareaX_H
#define TareaX_H //previene de incluir el módulo varias
//veces y generar error de duplicidad

//Definición de parámetros modificables del módulo
#define param1 VALOR1
#define param2 VALOR2
...
#define paramj VALORj

void TareaX_Init(void); //Constructor de la TareaX
void TareaX_Run(void); //Ejecución de la TareaX
void TareaX_Flush(void); //Destrucción de la TareaX
...
//otras funciones públicas

#endif
```

10.3.3 Archivo código fuente TareaX.C

El archivo con el código fuente del módulo denominado **TareaX.C** tiene la estructura que muestra el *Gráfico 10.8*.

**GRÁFICO
10.8**

Estructura general del archivo TareaX.C.

```
// Nombre del Módulo
//
//***** TareaX.C *****
//
// Fecha de Creación: MM DD AA
// Asunto: Explicación Resumida de la Función del Módulo
// Rev #

static char tareaXSt; //variable de estado

//Definición de los estados posibles de tareaXSt;
#define ESTADO_DUMMY 0      //estado de reposo
#define ESTADO_1      1
#define ESTADO_2      2
...
#define ESTADO_n      N

// Implementación de la función TareaX_Init
void TareaX_Init(void){
    ....
}
```

```
// Implementación de la función TareaX_Run
void TareaX_Run(void){
    switch(tareaXSt){
        case ESTADO_DUMMY:
            ...
            break;
        case ESTADO_1:
            ...
            break;
        case ESTADO_2:
            ...
            break;
        ...
        case ESTADO_n:
            ...
            break;
        default:
            ...
            break;
    }
}
// Implementación de la función TareaX_Flush
void TareaX_Flush(void){
    ...
}
...
//otras funciones públicas
...
//otras funciones privadas (static)
```



La función de
inicialización
TareaX_
Init(), inicializa el
hardware asociado

La variable de estado **tareaXSt**, por norma de declaración de variables, inicia con letra minúscula y además tiene el sufijo St, el cual indica que es una variable de estado, de esa forma el diseñador puede ubicar su naturaleza en el transcurso del programa sin necesidad de ir a su definición (sea un código desarrollado por él mismo o por otra persona del grupo de R&D).

al módulo, configura pines de entrada/salida y a los submódulos que contiene.



La variable de estado **St** deberá ser tipo **static char**, el cual previene que la variable sea accesada por un módulo externo que pueda modificar su valor, esto da mayor nivel de encapsulado al código y se garantiza que la variable solo es modificada en este módulo lo que permite encontrar con facilidad errores asociados al módulo. La dimensión char es suficiente para almacenar todos los estados posibles (256 estados), una tarea embebida que tenga más de 256 estados no se considera práctica y deberá ser convertida como mínimo a 10 sub-tareas (o módulos), además la variable char es la más eficiente en su manejo en arquitecturas de 8 bits.

La función de inicialización **TareaX_Init()**, deberá tener el sufijo **_Init**, para dar claridad al programador que ésta es una función de inicialización de un módulo. La función inicializa la variable **tareaXSt** con su valor inicial, en muchos casos al estado **ESTADO_DUMMY**, el cual hace que la tarea esté activa pero puede no realizar ninguna función hasta que no sucedan otros eventos. Esta función, además inicializa el hardware asociado al módulo, configura pines de entrada/salida y a los sub-módulos que contiene.

La función de ejecución de estado estable **TareaX_Run()**, está compuesta por una plantilla basada en la sentencia de control **switch**, con todos los estados (**case: break;**) posibles del módulo, puede incluirse una sección por defecto (**default**), que se ejecuta si el módulo no está en ninguno de los estados definidos (ver Gráfico 10.9).

Se recomienda poner todos los **case: break;** de cada uno de los estados del módulo, y solo al final de la depuración del código, si se identifican estados que realizan el mismo código, se colocan bajo el mismo **case**, y también si un estado tiene códigos comunes con otro estado se remueve el **break** correspondiente.

Es importante resaltar que el tiempo de ejecución que tarda el **switch** en determinar el valor de **tareaXSt** es el mismo, y no depende del estado en el que se encuentre el módulo.

Contrario a lo que sugiere el código y podría pensarse en una programación lineal, el preámbulo de ejecución de **TareaX_Run** es menor si se encuentra en los primeros estados: **ESTADO_1**, **ESTADO_2**, y mucho mayor para **ESTADO_n-1**, **ESTADO_n**.

Esta situación sería cierta si el compilador realiza la solución al **switch** comparando uno a uno los valores de forma consecutiva; por fortuna, muchos de los compiladores resuelven esta situación por medio de saltos en los cuales la constante de estado establece un *offset* al código que debe iniciar su ejecución.



**GRÁFICO
10.9**

Optimización de función TareaX_Run.

```
// Implementación de la función TareaX_Run
void TareaX_Run(void){
    switch(tareaXSt){
        case ESTADO_DUMMY:
            ...
        break;
        case ESTADO_1:
            ...
            //se remueve el break;
        case ESTADO_2:
            ...
            //código común a ESTADO_1 y
            //ESTADO_2
        break;
        case ESTADO_3:
            //ESTADO_3 y ESTADO_4
        case ESTADO_4:
            ...
            //ejecutan el mismo código
        ...
        break;

        ...
        case ESTADO_n:
            ...
        break;
        default:
            ...
        break;
    }
}
```

Aunque el código se vea muy extenso se debe recordar que en un llamado a **TareaX_Run**, únicamente se ejecuta un solo bloque **case break;** y se retorna el control al loop infinito **for(;;)** del programa principal.

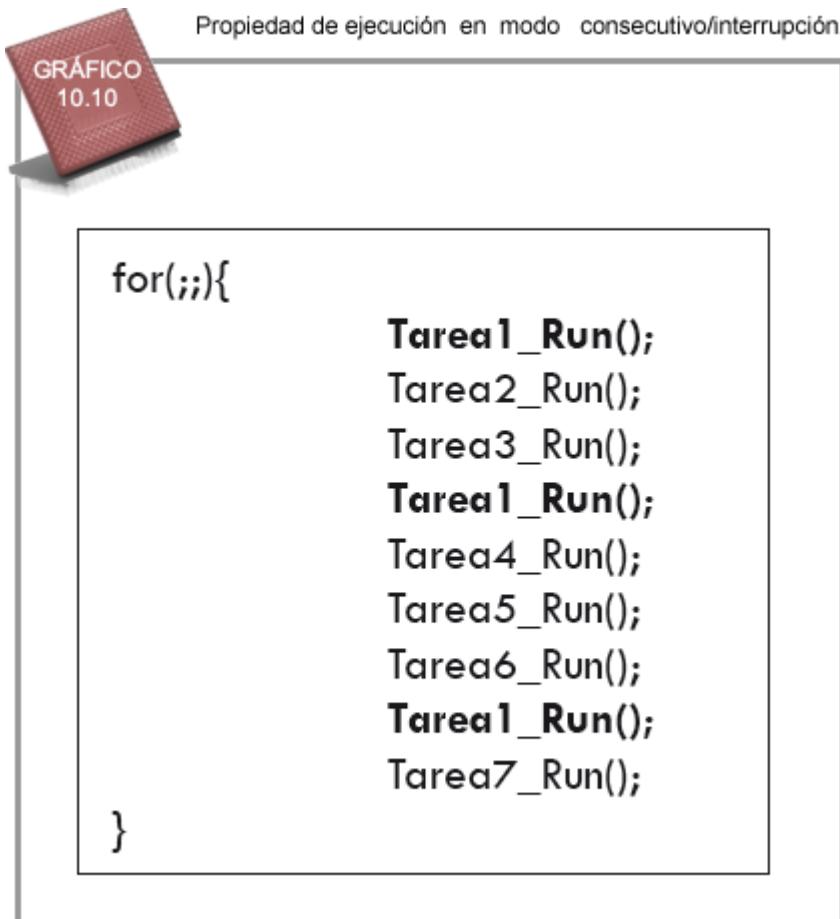
En algunos módulos se hace necesaria la implementación de la función *TareaX_Flush()*, destinada a suspender la ejecución de *TareaX_Run()* y ponerla en estado inactivo (**ESTADO_DUMMY**), lo mismo que desactivar el hardware asociado al módulo, ya sea porque el módulo no requiere ser ejecutado a partir de un punto o porque se requiere mayor tiempo del procesador para otra tareas.

10.3.4 Manejo de prioridades

En este esquema no se maneja la prioridad tal como se define para un sistema operativo; sin embargo, se puede hacer que la CPU del procesador tenga preferencia en la ejecución de una tarea de dos formas básicas:

La primera se hace al momento del diseño y se resuelve en tiempo de compilación, se recomienda cuando las tareas tienen prioridad constante o varía muy poco durante la ejecución del programa, consiste en adicionarla más de una vez en el Loop principal como lo ilustra el *Gráfico 10.10*.

De esta forma la CPU estará más pendiente de ejecutar la Tarea1 que las demás tareas, y su tiempo de respuesta será menor.



La segunda forma de manejárla es recomendada cuando la prioridad de una tarea es dinámica, es decir, puede cambiar a medida que el programa se ejecute y consiste en el manejo de una variable global que se puede declarar como:

unsigned char prioridadActual;

La cual es modificada por el módulo que desea elevar la prioridad y monitoreada por los demás módulos antes de iniciar la ejecución del switch general; si la variable **prioridadActual** es en valor menor que la prioridad definida para ella, significa que hay otras tareas que son de mayor prioridad y requieren mayor tiempo del procesador, en este caso el módulo obliga al retorno hasta que la prioridad baje nuevamente.

Las funciones **TareaX_Run()** se modifican como lo ilustra el *Gráfico 10.11*.

GRÁFICO
10.11

Manejo de prioridad dinámica en modo consecutivo/interrupción

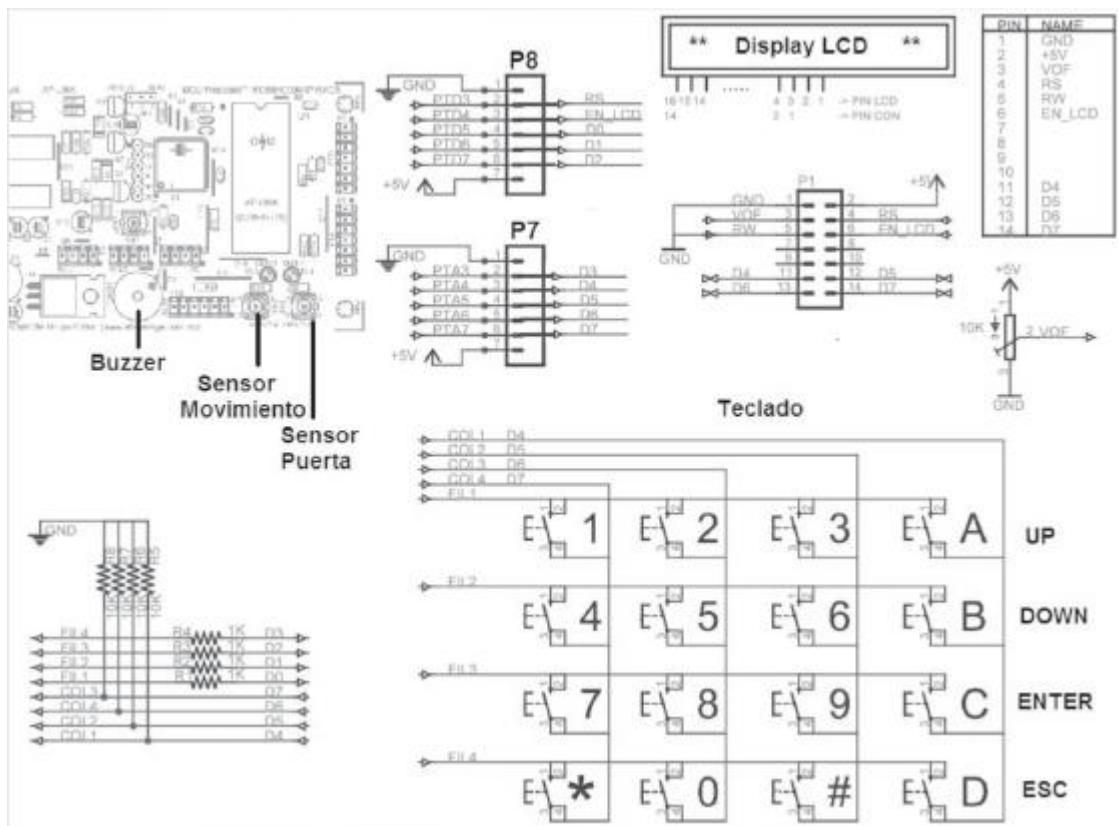
```
#define PRIORIDAD_X PRIO_X      //define prioridad inicial X
unsigned char prioridad_X=PRIO_X; //prioridad X en RAM
extern unsigned char prioridadActual; //variable global
void TareaX_Run(void){
    if(prioridadActual < prioridad_X) return;
    switch(tareaXSt){
        case ESTADO_DUMMY:
            ....
            break;
        case ESTADO_1:
            ...
            break;
        case ESTADO_2:
            ...
            break;
        case ESTADO_3:
            ...
            break;
        ...
        case ESTADO_n:
            ....
            break;
        default:
            ...
            break;
    }
}
```

EJEMPLO No. 35

Alarma de intrusión

Objetivo:

Usando el método consecutivo/interrupción diseñar un sistema de alarma de intrusión,el cual posee dos entradas: una usada como puerta principal y la otra como detector de movimiento, que cuente a sus vez con un display de LCD y un teclado matricial 4x4, que permiten respectivamente visualizar mensajes y facilitar la incorporación y parametrizacion del equipo. Ver grafico:



El sistema en estado estable (alarma armada) responde a la actividad de las dos señales de entrada, al detectar actividad solicita una clave de acceso para poder desactivar el paso a su estado de pánico.

Si la clave no es digitada en un tiempo determinado o es digitada de forma incorrecta, la alarma pasa al estado de pánico, el cual es notificado mediante una cadencia de una señal auditiva (buzzer).

El sistema debe mediante un menú permitir las siguientes configuraciones:

Fecha y hora, armar alarma, cambio de idioma, cambio de clave y prueba de sensores.

El sistema armado presenta la fecha y hora actual en la línea inferior del LCD, en la

superior se presentan las acciones que están teniendo lugar en la alarma como son actividad de los sensores.

Mientras no se tenga actividad presentar un mensaje especificado en forma secuencial en la línea superior.

Solución:

En el sistema se identifican las siguientes tareas:

Lcd: para el manejo del display, y el mensaje secuencial.

Key: para el manejo del teclado matricial.

Alarma: Módulo encargado del manejo de los estados de la alarma.

inOut: Módulo para el manejo de los sensores de entrada y las salidas.

Timer: Manejo del tiempo, fecha, temporizaciones.

Menu: manejo del menú de configuración.

Teniendo en cuenta que el módulo Timer funciona completamente por interrupción de una de las bases de tiempo no requiere ser procesado en el loop principal, como tampoco lo debe hacer para la tarea inOut, debido a que el manejo es solo de pines de entrada y salida y no tiene estados intermedios.

De esta forma el loop principal se verá como sigue:

```
/****** Ejemplo 35 *****
```

```

// Alarma Consecutivo/Interrupción

// Fecha: Abril 12,2009

// Asunto: Proyecto alarma de intrusión

// Hardware: Sistema de desarrollo AP-Link

// Versión: 1.0 Por: Gustavo A. Galeano A.

//*****



#include "includes.h"

void main(void){ /*Función Principal*/

    Mcu_Init(); //Inicializa módulo MCU

    InOut_Init(); //Inicializa módulo de InOut

    Lcd_Init(); //Inicializa módulo LCD

    Key_Init(); //Inicializa módulo Teclado

    Timer_Init(); //inicializa módulo TBM

    Menu_Init(); //Inicializa módulo Menú

    Alarma_Init(); //Inicializa módulo Alarma

    EnableInterrupts; /*enable interrupts */

    for(;;) {

        Key_Run(); //Ejecuta el módulo teclado

        Menu_Run(); //Ejecuta el módulo Menú

        Alarma_Run(); //Ejecuta el módulo Alarma

        Lcd_Run(); //Ejecuta el módulo LCD

        Mcu_Run(); //Ejecuta el módulo MCU

    } /* loop forever */

/* please make sure that you never leave main */

```

}

Los módulos que soportan el programa principal:

```
/* MENU.C */

struct tm *horaActualPtr; // Apuntador a la estructura de la hora
char bufferTime[5];

unsigned char idiomaSelected; //Almacenamiento de lenguaje actual
extern char pin_code[5];
extern time_t segAct;//Variable almacenamiento de los segundos desde 1 Enero 1970
const char *MenuMsgPtr[NRO_IDIOMAS][25]={

    "Ajuste Fecha/Hora?", "Armar Alarma?", "Prueba Sensores?",
    "Seleccion Idioma?", "Cambio de Clave?", "Seleccione Idioma",
    "Espanol ", "* Alarma Intrusion* ", " Menu Inicio ", "Ano> ",
    "Armando Alarma", "Entrar Clave: ", "Codigo Aceptado",
    "Codigo No Valido ", "Mes> ", "Dia> ", "Hora> ", "Min> ",
    "Seg> ", "Estado Sensores", "PIN Anterior:", "PIN Nuevo:",
    "Reentre PIN: ", "Pin Cambiado ", "** ALARMA VIOLADA ** ",
    "Set Date/Time?", "Arm Alarm?", "Auto Test?", "Select Language?",
    "Change PIN? ", "Select Language", "English", "*Intrusion Alarm* ",
    " Main Menu", "Year> ", "Alarm Armed", "Enter Key:", "Code Accepted",
    "Code Not Valid", "Month> ", "Day> ", "Hour> ", "Min> ", "Sec> ",
    "Sensor Status", "Old PIN: ", "New PIN: ", "Re-Enter PIN:", "Pin Was Changed",
    "*** ACTIVE ALARM ***",
```

```

};

static char menuSt; //variable de estado de la tarea Menú

enum{ //estados de la tarea Menú

    MENU_DUMMY=0, //inactivo

    MENU_TST_DT, //es fecha?

    MENU_GET_YEAR, //configuración del año

    MENU_GET_MONTH, //configuración del mes

    MENU_GET_DAY, //configuración del día

    MENU_GET_HOUR, //configuración de la hora

    MENU_GET_MIN, //configuración de minutos

    MENU_GET_SEC, //configuración de segundos

    MENU_TST_ARMADO, //armado de la alarma?

    MENU_TST_ATEST, //auto test de sensores?

    MENU_TST_LANG, //selección de lenguaje?

    MENU_TST_CH_PIN, //cambio de la clave?

    MENU_SEL_LANG, //selección de lenguaje?

    MENU_IN_ATEST, //en prueba de sensores

    MENU_IN_CH_PIN, //en cambio de pin

    MENU_IN_NEW_PIN, //solicitud de la nueva clave

    MENU_IN_RE_NEW_PIN, //solicitud de nueva clave nuevamente

    MENU_DISABLED, //menú inactivo

};

void Menu_Init(void){ //Función de inicialización de la Tarea Menú

    menuSt = MENU_DUMMY;
}

```

```

}

void Menu_Run(void){ //Función de ejecución de la sub-tarea Menú

char teclaMenu,dataOk=0;

static unsigned char idiomaSelectedTemp;

static int datoTemp;

static char indice,lastMovActivo,lastDoorActivo;

static char newData[5],newData2[5];

if(MenuActivo()){ GetTeclaValida(&teclaMenu);}

switch(menuSt){

case Menu==KEY_UP){

if(teclaMenu == KEY_UP){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][0]);

menuSt = MENU_TST_DT;

}

break;

case MENU_TST_DT:

if(teclaMenu == KEY_ENTER){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][9]);

segAct = GetTime();

horaActualPtr = localtime(&segAct);

datoTemp = horaActualPtr->tm_year + 1900;

(void)sprintf(bufferTime,"%u",datoTemp);

Lcd_Goto(26);

PutchD(bufferTime[0]);
}
}
}

```

```

        PutchD(bufferTime[1]);

        PutchD(bufferTime[2]);

        PutchD(bufferTime[3]);

        Lcd_Goto(33);

        indice = 0;

        menuSt = MENU_GET_YEAR;

    }

if(teclaMenu == KEY_UP){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][1]);

    menuSt = MENU_TST_ARMADO;

}

if(teclaMenu == KEY_DW){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][4]);

    menuSt = MENU_TST_CH_PIN;

}

break;

case MENU_TST_ARMADO:

if(teclaMenu == KEY_ENTER){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][10]);

    menuSt = MENU_DISABLED;

}

if(teclaMenu == KEY_UP){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][2]);

    menuSt = MENU_TST_ATEST;

```

```

    }

if(teclaMenu == KEY_DW){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][0]);

    menuSt = MENU_TST_DT;

}

break;

case MENU_GET_YEAR:

if(teclaMenu == KEY_ENTER){

    if(indice == 4){ //se entro un anyo de 4 digitos

        newData[4] = 0;

        datoTemp = atoi(newData);

        horaActualPtr->tm_year = datoTemp - 1900;

        segAct = mktime(horaActualPtr);

    }

    dataOk = 1;

}else{

    if(isdigit(teclaMenu)){

        if(indice <4){

            newData[indice] = teclaMenu;

            PutchD(teclaMenu);

            indice++;

        }else{

            newData[4] = 0;

            datoTemp = atoi(newData);

        }

    }

}

```

```

        horaActualPtr->tm_year = datoTemp - 1900;

        segAct = mktime(horaActualPtr);

        dataOk = 1;

    }

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

if(dataOk){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][14]);

    horaActualPtr = localtime(&segAct);

    datoTemp = horaActualPtr->tm_mon+1;

    (void)sprintf(bufferTime,"%u",datoTemp);

    Lcd_Goto(26);

    if(datoTemp > 9){

        PutchD(bufferTime[0]);

        PutchD(bufferTime[1]);

    }else{

        PutchD('0');

        PutchD(bufferTime[0]);

    }

    Lcd_Goto(33);

    indice = 0;

    menuSt = MENU_GET_MONTH;

}

```

```

break;

case MENU_GET_MONTH:

    if(teclaMenu == KEY_ENTER){

        if(indice == 2){ //entraron un mes

            newData[2] = 0;

            datoTemp = atoi(newData)-1;

            horaActualPtr->tm_mon = datoTemp;

            segAct = mktime(horaActualPtr);

        }

        dataOk = 1;

    }else{

        if(isdigit(teclaMenu)){

            if(indice <2){

                newData[indice] = teclaMenu;

                PutchD(teclaMenu);

                indice++;

            }else{

                newData[2] = 0;

                datoTemp = atoi(newData)-1;

                horaActualPtr->tm_mday = datoTemp;

                segAct = mktime(horaActualPtr);

                dataOk = 1;

            }

        }

    }

}

```

```

    }

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

if(dataOk){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][15]);

    horaActualPtr = localtime(&segAct);

    datoTemp = horaActualPtr->tm_mday;

    (void)sprintf(bufferTime,"%u",datoTemp);

    Lcd_Goto(26);

    if(datoTemp > 9){

        PutchD(bufferTime[0]);

        PutchD(bufferTime[1]);

    }else{

        PutchD('0');

        PutchD(bufferTime[0]);

    }

    Lcd_Goto(33);

    indice = 0;

    menuSt = MENU_GET_DAY;

}

break;

case MENU_GET_DAY:

    if(teclaMenu == KEY_ENTER){

        if(indice == 2){

            newData[2] = 0;

```

```

datoTemp = atoi(newData);

horaActualPtr->tm_mday = datoTemp;

segAct = mktime(horaActualPtr);

}

dataOk = 1;

}else{

if(isdigit(teclaMenu)){

if(indice <2){

newData[indice] = teclaMenu;

PutchD(teclaMenu);

indice++;

}else{

newData[2] = 0;

datoTemp = atoi(newData);

horaActualPtr->tm_mday = datoTemp;

segAct = mktime(horaActualPtr);

dataOk = 1;

}

}

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

if(dataOk){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][16]);

horaActualPtr = localtime(&segAct);

```

```

datoTemp = horaActualPtr->tm_hour;

(void)sprintf(bufferTime,"%u",datoTemp);

Lcd_Goto(26);

if(datoTemp > 9){

    PutchD(bufferTime[0]);

    PutchD(bufferTime[1]);

}else{

    PutchD('0');

    PutchD(bufferTime[0]);

}

Lcd_Goto(33);

indice = 0;

menuSt = MENU_GET_HOUR;

}

break;

case MENU_GET_HOUR:

if(teclaMenu == KEY_ENTER){

    if(indice == 2){

        newData[2] = 0;

        datoTemp = atoi(newData);

        horaActualPtr->tm_hour = datoTemp;

        segAct = mktime(horaActualPtr);

    }

    dataOk = 1;
}

```

```

}else{

    if(isdigit(teclaMenu)){

        if(indice <2){

            newData[indice] = teclaMenu;

            PutchD(teclaMenu);

            indice++;

        }else{

            newData[2] = 0;

            datoTemp = atoi(newData);

            horaActualPtr->tm_hour = datoTemp;

            segAct = mktime(horaActualPtr);

            dataOk = 1;

        }

    }

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

if(dataOk){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][17]);

    horaActualPtr = localtime(&segAct);

    datoTemp = horaActualPtr->tm_min;

    (void)sprintf(bufferTime,"%u",datoTemp);

    Lcd_Goto(26);

    if(datoTemp > 9){

        PutchD(bufferTime[0]);
}

```

```

    PutchD(bufferTime[1]);

}else{
    PutchD('0');

    PutchD(bufferTime[0]);

}

Lcd_Goto(33);

indice = 0;

menuSt = MENU_GET_MIN;

}

break;

case MENU_GET_MIN:

if(teclaMenu == KEY_ENTER){

    if(indice == 2){

        newData[2] = 0;

        datoTemp = atoi(newData);

        horaActualPtr->tm_min = datoTemp;

        segAct = mktime(horaActualPtr);

    }

    dataOk = 1;

}else{

    if(isdigit(teclaMenu)){

        if(indice <2){

            newData[indice] = teclaMenu;

            PutchD(teclaMenu);

        }

    }

}

```

```

        indice++;

    }else{

        newData[2] = 0;

        datoTemp = atoi(newData);

        horaActualPtr->tm_min = datoTemp;

        segAct = mktime(horaActualPtr);

        dataOk = 1;

    }

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

if(dataOk){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][18]);

    horaActualPtr = localtime(&segAct);

    datoTemp = horaActualPtr->tm_sec;

    (void)sprintf(bufferTime,"%u",datoTemp);

    Lcd_Goto(26);

    if(datoTemp > 9){

        PutchD(bufferTime[0]);

        PutchD(bufferTime[1]);

    }else{

        PutchD('0');

        PutchD(bufferTime[0]);

    }

}

```

```

    Lcd_Goto(33);

    indice = 0;

    menuSt = MENU_GET_SEC;

}

break;

case MENU_GET_SEC:

if(teclaMenu == KEY_ENTER){

    if(indice == 2){

        newData[2] = 0;

        datoTemp = atoi(newData);

        horaActualPtr->tm_sec = datoTemp;

        segAct = mktime(horaActualPtr);

    }

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][1]);

    menuSt = MENU_TST_ARMADO;

}else{

    if(isdigit(teclaMenu)){

        if(indice <2){

            newData[indice] = teclaMenu;

            PutchD(teclaMenu);

            indice++;

        }else{

            newData[2] = 0;

            datoTemp = atoi(newData);

        }

    }

}

```

```

horaActualPtr->tm_sec = datoTemp;

segAct = mktime(horaActualPtr);

menuSt = MENU_DUMMY;

}

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

break;

case MENU_TST_ATEST:

if(teclaMenu == KEY_UP){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][3]);

menuSt = MENU_TST_LANG;

}

if(teclaMenu == KEY_DW){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][1]);

menuSt = MENU_TST_ARMADO;

}

if(teclaMenu == KEY_ENTER){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][19]);

menuSt = MENU_IN_ATEST;

}

break;

case MENU_IN_ATEST:

if(teclaMenu == KEY_ESC){

```

```

BuzzerOFF();

PrintLCD2nd(MenuMsgPtr[idiomaSelected][3]);

menuSt = MENU_TST_LANG;

}

if(Sensor_Movimiento_Activo() && (!lastMovActivo)){

    lastMovActivo = TRUE;

    Lcd_Goto(35);

    PutchD('X');

    BuzzerON();

}

if(!Sensor_Movimiento_Activo() && lastMovActivo){

    lastMovActivo = FALSE;

    Lcd_Goto(35);

    PutchD('O');

    BuzzerOFF();

}

if(Sensor_Door_Activo() && (!lastDoorActivo)){

    lastDoorActivo = TRUE;

    Lcd_Goto(37);

    PutchD('X');

    BuzzerON();

}

if(!Sensor_Door_Activo() && lastDoorActivo){

    lastDoorActivo = FALSE;

```

```

    Lcd_Goto(37);

    PutchD('O');

    BuzzerOFF();

}

break;

case MENU_TST_LANG:

    if(teclaMenu == KEY_UP){

        PrintLCD2nd(MenuMsgPtr[idiomaSelected][4]);

        menuSt = MENU_TST_CH_PIN;

    }

    if(teclaMenu == KEY_DW){

        PrintLCD2nd(MenuMsgPtr[idiomaSelected][2]);

        menuSt = MENU_TST_ATEST;

    }

    if(teclaMenu == KEY_ENTER){

        Lcd_Goto(0);

        PutsD((unsigned char *)MenuMsgPtr[idiomaSelected][5]);

        PrintLCD2nd(MenuMsgPtr[idiomaSelected][6]);

        idiomaSelectedTemp = idiomaSelected;

        menuSt = MENU_SEL_LANG;

    }

break;

case MENU_SEL_LANG:

    if(teclaMenu == KEY_UP){


```

```

idiomaSelectedTemp = (unsigned char)(idiomaSelectedTemp+1)%NRO_IDIOMAS;
PrintLCD2nd(MenuMsgPtr[idiomaSelectedTemp][6]);
}

if(teclaMenu == KEY_DW){

idiomaSelectedTemp = (unsigned char)(idiomaSelectedTemp-1)%NRO_IDIOMAS;
PrintLCD2nd(MenuMsgPtr[idiomaSelectedTemp][6]);
}

if(teclaMenu == KEY_ENTER){

Lcd_Goto(0);

idiomaSelected = idiomaSelectedTemp;

PutsD((unsigned char *)MenuMsgPtr[idiomaSelected][7]);

PrintLCD2nd(MenuMsgPtr[idiomaSelected][4]);

menuSt = MENU_TST_CH_PIN;

}

break;

case MENU_TST_CH_PIN:

if(teclaMenu == KEY_UP){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][0]);

menuSt = MENU_TST_DT;

}

if(teclaMenu == KEY_DW){

PrintLCD2nd(MenuMsgPtr[idiomaSelected][3]);

menuSt = MENU_TST_LANG;

}

```

```

if(teclaMenu == KEY_ENTER){

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][20]);

    indice = 0;

    Lcd_Goto(33);

    menuSt = MENU_IN_CH_PIN;

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

break;

case MENU_IN_CH_PIN:

if(teclaMenu == KEY_ENTER){

    if(indice == 4){

        if((pin_code[0]==newData[0])&&(pin_code[1]==newData[1])&&(pin_
code[2]==newData[2])&&(pin_code[3]==newData[3])){

            PrintLCD2nd(MenuMsgPtr[idiomaSelected][21]);

            indice = 0;

            Lcd_Goto(33);

            menuSt = MENU_IN_NEW_PIN;

        }else{

            PrintLCD2nd("Error PIN");

            menuSt = MENU_DUMMY;

        }

    }

}

if(isdigit(teclaMenu)){


```

```

if(indice <4){

    newData[indice] = teclaMenu;

    PutchD('*');

    indice++;

}

}else{

    menuSt = MENU_DUMMY;

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

break;

case MENU_IN_NEW_PIN:

    if(teclaMenu == KEY_ENTER){

        if(indice == 4){

            PrintLCD2nd(MenuMsgPtr[idiomaSelected][22]);

            indice = 0;

            Lcd_Goto(33);

            menuSt = MENU_IN_RE_NEW_PIN;

        }

    }

    if(isdigit(teclaMenu)){

        if(indice <4){

            newData[indice] = teclaMenu;

            PutchD('*');

            indice++;

        }

    }

}

```

```

}else{
    menuSt = MENU_DUMMY;
}

}

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

break;

case MENU_IN_RE_NEW_PIN:
    if(teclaMenu == KEY_ENTER){
        if((indice==4)&&(newData[0]==newData2[0])&&
           (newData[1]==newData2[1])&&(newData[2]==newData2[2])&&(newData[3]
==newData2[3])){
            pin_code[0] = newData[0]; pin_code[1] = newData[1];
            pin_code[2] = newData[2]; pin_code[3] = newData[3];
            PrintLCD2nd(MenuMsgPtr[idiomaSelected][23]);
            menuSt = MENU_DUMMY;
        }
    }
    if(isdigit(teclaMenu)){
        if(indice <4){
            newData2[indice] = teclaMenu;
            PutchD('*');
            indice++;
        }else{ menuSt = MENU_DUMMY; }
    }
}

```

```

if(teclaMenu == KEY_ESC){ menuSt = MENU_DUMMY; }

break;

case MENU_DISABLED:

break;

}

}

char MenuActivo(void){

    return (menuSt != MENU_DISABLED);

}

/* inOut.C */

#include "includes.h"

// Funciones de encendido y apagado de LEDs

void Led_Neg(void){

    DDRC_DDRC3 = 1; PTC_PTC3 ^= 1;

}

void Led_Out2_ON(void){

    DDRC_DDRC2 = 1; PTC_PTC2 = 1;

}

void Led_Out2_OFF(void){

    DDRC_DDRC2 = 1; PTC_PTC2 = 0;

}

void BuzzerON(void){

    DDRD_DDRD0 = 1; PTD_PTD0 = 1;

}

```

```

void BuzzerOFF(void){

    DDRD_DDRD0 = 1; PTD_PTD0 = 0;

}

void InOut_Init(void){

    BuzzerOFF();

    Led_Out2_OFF();

}

/* ALARMA.C */

#include "includes.h"

char pin_code[5]; /*variable código desactivado de la alarma*/

extern unsigned char idiomaSelected; /*vble de almacenamiento del lenguaje
actual*/

extern const char *MenuMsgPtr[NRO_IDIOMAS][25]; /*mensajes en el LCD en
todos los idiomas*/

extern struct tm *horaActualPtr; /*apuntador a la estructura de la hora*/

char alarmaSt; /*variable de estado de la tarea Alarma*/

enum{ /*estados de la tarea Alarma*/

    ALARMA_DUMMY=0, //inactivo

    ALARMA_MENU, //en ejecución del Menú

    ALARMA_WT_TIME_ARMADO, //esperando tiempo para armarse

    ALARMA_ARMADA, //armada verificando sensores

    ALARMA_WT_CODE, //esperando el código ***

    ALARMA_CODE_INVALID, //el código fue inválido
}

```

```

ALARMA_CODE_VALID, //el código fue valido

ALARMA_BUZZER_ON_WT_OFF, //buzzerON, esperando para apagar

ALARMA_BUZZER_OFF_WT_ON, //buzzerOFF, esperando para encender

};

void Alarma_Init(void){ //Función de Inicialización de la tarea Alarma

    alarmaSt = ALARMA_DUMMY;

    PutsD((unsigned char *)MenuMsgPtr[idiomaSelected][7]);

    PrintLCD2nd(MenuMsgPtr[idiomaSelected][8]);

    pin_code[0]=PIN_DEFAULT[0]; //inicialización con el código por defecto

    pin_code[1]=PIN_DEFAULT[1];

    pin_code[2]=PIN_DEFAULT[2]; pin_code[3] = PIN_DEFAULT[3];

}

void Alarma_Run(void){ //Función de Ejecución de la tarea Alarma

char teclaPresionada; //almacenamiento temporal de la teclaPresionada

static unsigned char cntCode,code[4];

static time_t timeOutAlarma,horaActualSeg;

switch(alarmaSt){

    case ALARMA_DUMMY:

    case ALARMA_MENU:

        if(MenuActivo() == FALSE){

            alarmaSt = ALARMA_WT_TIME_ARMADO;

            timeOutAlarma = GetTime() + TIME_TO_WAIT_ARMADO;

            (void)TimeChange();

            Lcd_StartPrint();

```

```

    }

break;

case ALARMA_WT_TIME_ARMADO:

    GetTeclaValida(&teclaPresionada);

    if(GetTime() & 0x01){BuzzerOFF();}else{BuzzerON();}

    if(TestTime(timeOutAlarma)){

        BuzzerOFF();

        alarmaSt = ALARMA_ARMADA;

    }

break;

case ALARMA_ARMADA:

    GetTeclaValida(&teclaPresionada);

    if(Sensor_Door_Activo() || Sensor_Movimiento_Activo() ||

(teclaPresionada == KEY_UP)) {

        BuzzerOFF();

        PrintLCD2nd(MenuMsgPtr[idiomaSelected][11]);

        cntCode = 0;

        Lcd_Goto(33);

        timeOutAlarma = GetTime() + TIME_TO_ENTER_CODE;

        alarmaSt = ALARMA_WT_CODE;

    }

if(TimeChange()){ //cambiaron los minutos?

    horaActualSeg= GetTime();

    horaActualPtr = localtime(&horaActualSeg);

```

```

    PrintLCD2nd(asctime(horaActualPtr));

}

break;

case ALARMA_WT_CODE:

    GetTeclaValida(&teclaPresionada);

    if((teclaPresionada && (cntCode<4))){
        code[cntCode] = teclaPresionada;
        cntCode++;
        Lcd_Goto(33+cntCode);
        PutchD('*');
    }

    if(TestTime(timeOutAlarma)){ alarmaSt = ALARMA_CODE_INVALID; }

    if(cntCode ==4 ){
        code[cntCode] = 0;
        if(!strcmp(code,pin_code)){ //codigo entrado igual al esperado?
            Lcd_Flush();
            PrintLCD2nd(MenuMsgPtr[idiomaSelected][12]);
            alarmaSt = ALARMA_CODE_VALID;
        }else{
            PrintLCD2nd(MenuMsgPtr[idiomaSelected][13]);
            alarmaSt = ALARMA_CODE_INVALID;
        }
    }

break;

```

```

case ALARMA_CODE_INVALID:

    Lcd_Flush(); //suspende manejo en la primera fila

    Lcd_Goto(0);

    PutsD((unsigned char*)MenuMsgPtr[idiomaSelected][24]);

    BuzzerON();

    timeOutAlarma = GetTime() + TIME_BUZZER_ON;

    alarmaSt = ALARMA_BUZZER_ON_WT_OFF;

    break;

case ALARMA_BUZZER_ON_WT_OFF:

    GetTeclaValida(&teclaPresionada);

    if(TestTime(timeOutAlarma)) {

        BuzzerOFF();

        timeOutAlarma = GetTime() + TIME_BUZZER_OFF;

        alarmaSt = ALARMA_BUZZER_OFF_WT_ON;

    }

    if(teclaPresionada == KEY_UP) {

        BuzzerOFF();

        Menu_Init();

        alarmaSt = ALARMA_MENU;

    }

    break;

case ALARMA_BUZZER_OFF_WT_ON:

    GetTeclaValida(&teclaPresionada);

    if(TestTime(timeOutAlarma)) {

```

```

        BuzzerON();

        timeOutAlarma = GetTime() + TIME_BUZZER_OFF;

        alarmaSt = ALARMA_BUZZER_ON_WT_OFF;

    }

    if(teclaPresionada == KEY_UP){

        BuzzerOFF();

        Menu_Init();

        alarmaSt = ALARMA_MENU;

    }

    break;

case ALARMA_CODE_VALID:

    GetTeclaValida(&teclaPresionada);

    if(teclaPresionada == KEY_UP){

        BuzzerOFF();

        Menu_Init();

        alarmaSt = ALARMA_MENU;

    }

    break;

}

/* KEY.C */

#include "derivative.h" /* include peripheral declarations */

#include "key.h"

#define FILA1_OFF() DDRD_DDRD5 = 0

```

```

#define FILA2_OFF() DDRD_DDRD6 = 0

#define FILA3_OFF() DDRD_DDRD7 = 0

#define FILA4_OFF() DDRA_DDRA3 = 0

#define COL1() PTA_PTA4

#define COL2() PTA_PTA5

#define COL3() PTA_PTA6

#define COL4() PTA_PTA7

#define NRO_FILAS 4

#define NRO_COLUMNAS 4

char teclaPress,keySt; //variable de estados de teclado

#define KEY_DUMMY 0

#define KEY_TST_Fi 1

#define KEY_WAIT_F 5

const char teclado[NRO_FILAS][NRO_COLUMNAS]={ //Teclado Matricial 4x4

    '1','2','3','A',
    '4','5','6','B',
    '7','8','9','C',
    '*', '0', '#', 'D',
};

#define ENABLE_FILA_1() PTD_PTD5=1; DDRD_DDRD5 = 1 //Definición Filas y Columnas

#define ENABLE_FILA_2() PTD_PTD6=1; DDRD_DDRD6 = 1

#define ENABLE_FILA_3() PTD_PTD7=1; DDRD_DDRD7 = 1

#define ENABLE_FILA_4() DDRA_DDRA3 = 1; PTA_PTA3 = 1

static void Fila1_ON(void){ ENABLE_FILA_1(); }

```

```
static void Fila2_ON(void){ ENABLE_FILA_2(); }

static void Fila3_ON(void){ ENABLE_FILA_3(); }

static void Fila4_ON(void){ ENABLE_FILA_4(); }

static void Enable_Cols(){

    DDRA_DDRA4 = 0; DDRA_DDRA5 = 0;

    DDRA_DDRA6 = 0; DDRA_DDRA7 = 0;

}

const vFuncPtrV FilasPtr[NRO_FILAS]={

    Fila1_ON, Fila2_ON, Fila3_ON, Fila4_ON

};

static void Filas_OFF(void){

    Enable_Cols();

    FILA1_OFF();

    FILA2_OFF();

    FILA3_OFF();

    FILA4_OFF();

}

static void Filas_ON(void){

    ENABLE_FILA_1();

    ENABLE_FILA_2();

    ENABLE_FILA_3();

    ENABLE_FILA_4();

}

static char Test_Cols(void){
```

```

if(COL1()) return 4;

if(COL2()) return 3;

if(COL3()) return 2;

if(COL4()) return 1;

return 0;

}

void Key_Init(void){ //Función de Inicialización de Teclado

keySt = KEY_TST_Fi;

}

void Key_Run(void){ // Función de Ejecución de Teclado

static char nroFilaTst=0;

Filas_OFF();

switch(keySt){

    case KEY_DUMMY:

        break;

    case KEY_TST_Fi:

        FilasPtr[nroFilaTst]();

        if(Test_Cols()){

            teclaPress = teclado[nroFilaTst][Test_Cols()-1];

            keySt = KEY_WAIT_F;

        }

        nroFilaTst++;

        nroFilaTst %=NRO_FILAS;

        break;

}

```

```

case KEY_WAIT_F:

    Filas_ON();

    if(!Test_Cols()){

        keySt = KEY_TST_Fi;

    }

    break;

}

Filas_OFF();

}

static char TeclaValida(void){/*Indica si una tecla fue presionada*/


    return teclaPress;

}

void GetTeclaValida(char *teclaPtr){/*Retorna el Ascii de la tecla*/


    if(TeclaValida()){ *teclaPtr = teclaPress; }else{ *teclaPtr =0; }

    teclaPress = 0;

}

/* CPU.C */

#include "includes.h"

void Mcu_Init(void){ /*Función de Inicialización del registros de MCU*/

    CONFIG1 =


        (CONFIG1_LVIRSTD_MASK|CONFIG1_LVIPWRD_MASK|


        CONFIG1_LVIPWRD_MASK|CONFIG1_COPD_MASK|CONFIG1_STOP_MASK);

    CONFIG2 =


        (CONFIG2_STOP_ICLKDIS_MASK|CONFIG2_OSCCLK1_MASK|

```

```

CONFIG2_STOP_XCLKEN_MASK | CONFIG2_SCIBDSRC_MASK);

//habilita oscilador en modo STOP y SCIBDSRC

}

void Mcu_Run(void){

    LowPowerWait(); //Modo de bajo consumo

}

```

Función adicional del módulo LCD .C:

```

void Lcd_Run(void){

    static unsigned char indexMsg=0;

    unsigned char i;

    switch(lcdSt){

        case LCD_WAIT_TIME:

            if(TestTime(lcdTimer)){

                for(i=0;i<20;i++){

                    bufferMsg[i] = msgToPrint[indexMsg+i];

                }

                bufferMsg[20] = 0;

                indexMsg = (indexMsg + 1)%sizeof(msgToPrint);

                lcdSt = LCD_PRINT_MSG;

            }

            break;

        case LCD_PRINT_MSG:

```

```
Lcd_Goto(0);

PutsD(bufferMsg);

lcdTimer = GetTime() + TIME_TO_PRINT_MSG;

lcdSt = LCD_WAIT_TIME;

break;

}

}

void Lcd_Flush(void){

lcdSt = LCD_DUMMY;

}
```

Discusión:

Inicialmente se realiza una lista de los módulos que incorporará el sistema, el archivo main.c es usado como plantilla para realizar las pruebas de ciertas funciones de los módulos, así se realiza una prueba sobre las funciones del LCD y luego se guardan las funciones que pertenecen a este módulo en un archivo nuevo llamado LCD.C, y se crea un nuevo archivo llamado LCD.H que contiene la lista de las funciones públicas que serán accedidas por los módulos externos.

Nuevamente se usa el archivo main.c como plantilla para el manejo del teclado matricial, se realiza un pequeño programa de prueba que solo realiza lectura del teclado y lo imprime en el display LCD, esto para ensayar el correcto funcionamiento y

verificar que no existen conflictos en el manejo multiplexado del lcd y del teclado.

Se modulariza luego el módulo KEY.C y KEY.H.

Se realiza el módulo ALARMA.C en el cual se incluyen las labores del menú de configuración, sin embargo al realizarlo se nota que los estados del menú son varios por lo que se decide crear un nuevo modulo llamado MENU, que resuelve solo la sección del manejo del menú usando las teclas UP, DOWN, ENTER y ESC.

Se ensaya este módulo y se modulariza el un archivo separado llamados MENU.H y MENU.C.

Materiales adicionales en la



Alarma de intrusión:
proyecto completo para
Freescale™ (AP-Link) y
Microchip™ (PIC-Link).

Por último, se crean algunos módulos de apoyo como son el de INOUT,C que permite el manejo de los sensores de entrada y el buzzer y un módulo

llamado MCU.C que tiene macros para el manejo de zonas críticas y una función que realiza llamado a la instrucción de bajo consumo WAIT.

Es de notar que todos los módulos tienen una variable de estado tareas y se crea una lista enumerada enum de los estados que puede contener ese módulo, se crea la secuencia switch y se soluciona cada uno de los estados de forma independiente; esta técnica permite que si en algún momento se modifica la especificación de la alarma, se puede ubicar de forma directa la sección de código que requiere cambio, y se adicionan estados o se comprueban estados redundantes.

10.4 SISTEMA RTOS UC/OS-II DE MICRIUM

Es un sistema operativo de tiempo real preemptivo robusto y seguro escrito en lenguaje C, con una porción de lenguaje ensamblador para adaptarlo a diferentes arquitecturas y que ha sido certificado, incluso, para aplicaciones médicas y de aeronáutica (FAA¹).

El código fuente está 100% disponible para su uso en la arquitectura seleccionada, además de ofrecer un ajuste a la aplicación, permitiendo usar o desistir de algunos de los recursos del μC/OS-II que posibilitan ajustar el tamaño de memoria de programa (Flash) y memoria RAM adecuada a las limitaciones del procesador.

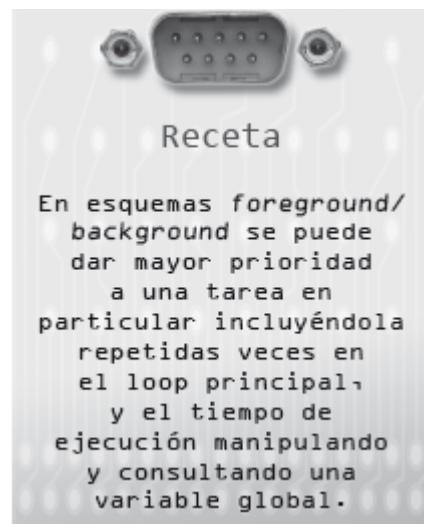
Este RTOS ha ganado una gran cantidad de usuarios alrededor del mundo, muchas instituciones educativas usan el μC/OS-II dentro de sus programas para la enseñanza de sistemas de tiempo real y numerosas compañías lo utilizan en el desarrollo de cámaras digitales, equipo de audio de alto desempeño, instrumentos médicos, instrumentos musicales, control de motores, cajeros automáticos, robots industriales, entre otros.

El sistema μC/OS-II puede ser usado con propósitos educativos sin pagar ninguna licencia, por lo que resulta apropiado para el propósito de este capítulo; sin embargo, para uso comercial requiere ser licenciado tal cual como se discute en el Apéndice B “Licensing Policy for μC/OS-II” del libro MicroC/OS-II The Real Time Kernel 2nd Edition, o bien en la página www.micrium.com.

Una de las ventajas radica en el hecho que el μC/OS-II ha sido ajustado a más de 40 arquitecturas de diferente marca que van desde los 8 hasta los 64 bits, incluyendo los DSP, lo que proporciona una permanencia al código desarrollado independiente de la máquina seleccionada o la vigencia de ésta en el mercado. El número de tareas máxima es de 56 con prioridad única de 64 niveles, lo que significa que su *scheduler* no permite el manejo *round-robin*.

Este sistema operativo ha sido escrito de forma tal que gran parte del código es portable a cualquier microcontrolador con manejo de stack y que sus registros de trabajo puedan ser almacenados cuando se presenta una interrupción.

Está compuesto de varios archivos con extensión .C .H y .asm. La particularización a una u otra arquitectura se realiza sustituyendo una sección del sistema llamada **port**, en la cual están los procedimientos específicos de manejo de interrupciones.



En esquemas foreground/background se puede dar mayor prioridad a una tarea en particular incluyéndola repetidas veces en el loop principal, y el tiempo de ejecución manipulando y consultando una variable global.



El sistema μC/OS-II ha ganado mucha aceptación debido a su robustez, la posibilidad de adaptarse a muchas arquitecturas y al hecho de que no es necesario pagar ninguna licencia por su uso para fines educativos.

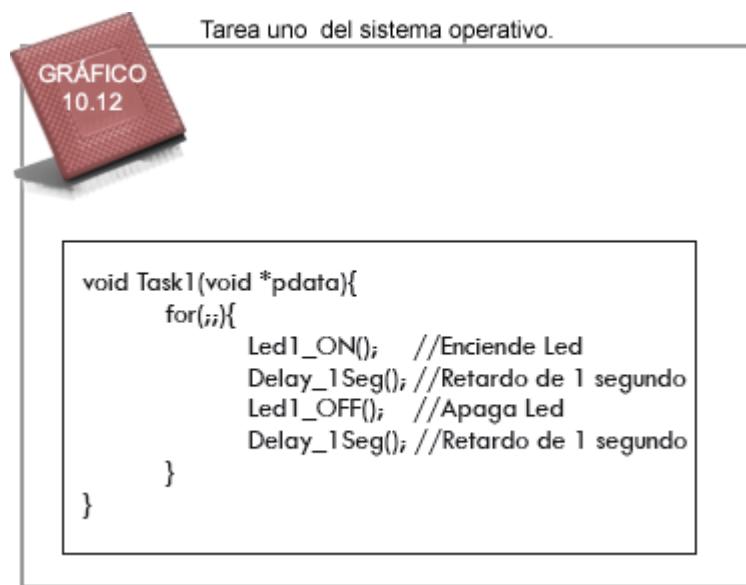


El diseñador de la aplicación final debe incorporar en su proyecto los folders y archivos requeridos por el µC/OS-II y crear las tareas que su aplicación exige.

1 FAA:(Federal Aviation Administration) www.faa.gov

10.4.1 Consideración para la implementación del uC/OS-II

Al cargar el proyecto maestro, el programador se ubica en el **main()**, en el cual se implementará una función cualquiera, en este caso será la ilustrada en el *Gráfico 10.12*, que enciende y apaga un LED a intervalos de un segundo:



El sistema µC/OS-II requiere de algunas configuraciones iniciales, como el número de tareas que el diseñador va a incorporar y el tamaño del stack para cada una de ellas.

La función básica que se encarga de realizar la creación y posterior llamado de una tarea en el µC/OS-II es **OSTaskCreate()**,



El sistema µC/OS-II requiere de algunas configuraciones iniciales, como es el número de tareas que el diseñador va a incorporar (**N_TASKS**) mediante la definición de un macro, en este caso se definirán como máximo dos (2) tareas.

También se requiere definir el tamaño del stack requerido por cada tarea, en este caso se definirá como un valor constante de 20, aunque el ejemplo es sencillo, este valor es suficiente para soportarlo.

También el sistema exige ser inicializado antes de arrancar con la creación de tareas, esto se realiza invocando la función **OSInit()**, donde ya se pueden empezar a incorporar las tareas.

Las tareas se incorporan al sistema operativo mediante la función **OSTaskCreate()** que tiene el siguiente prototipo:

```
INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);
```

El argumento # 1 *task* especifica el apuntador a la tarea que debe ser creada, en este caso **Task1**; el argumento # 2 *pdata* precisa un apuntador al argumento que debe ser entregado a la tarea, una vez que la tarea es llamada por el *scheduler*, en este caso la función no recibe argumento alguno, por lo que acá el argumento será NULL ((void*)0); el argumento # 3 *ptos*, se refiere a un apuntador al tope del stack que va a ser asignado a la tarea, y el argumento # 4 *prio* designa a la prioridad que se quiere para esta tarea, en este caso y por ser la única tarea se le asignará la máxima prioridad: **0** (cero).

Una vez creada al menos una tarea el sistema puede arrancar su ejecución, la cual se realiza mediante el llamado de la función **OSStart()**;

Las funciones de retardo están facilitadas por el sistema operativo y son las que ayudan a que la tarea pase del estado **Run** al estado **Wait**, sin esto el sistema quedará ocupado y no podrá seguir ejecutando otra tareas, para este caso particular de retardo de 1 segundo se acude a la función implementada del µC/OS-II llamada **OSTimeDlyHMSM(0,0,1,0)**; así la implementación de **Delay_1Seg()** es:

```
#define Delay_1Seg() OSTimeDlyHMSM(0,0,1,0);
```

El cuerpo del primer programa usando µC/OS-II se verá como lo ilustra el *Gráfico 10.13* a continuación:

```
// Definiciones para el µC/OS-II
#define TASK_STK_SIZE      20 //Define tamaño del stack de tareas
#define N_TASKS             2  //Número de tareas máximo

OS_STK TaskStk[N_TASKS][TASK_STK_SIZE];
OS_STK TaskStartStk[TASK_STK_SIZE];
char   TaskData[N_TASKS];

/* ***** Programa Principal **** */
void main(void){
    OSInit();
    OSTaskCreate(Task1,(void *)0, &TaskStartStk[TASK_STK_SIZE-1],0);
    OSStart();
}
```

Luego, si la tarea quiere removverse de la lista de tareas que ejecuta el *scheduler*, puede hacerse mediante la función **OSTackDel()**, la cual recibe como argumento la prioridad de la tarea que quiere removverse, de tal forma que la tarea creada anteriormente puede ser removida con el código:

OSTaskDel(0);

µC/OS-II siempre crea una tarea nula (*idle task*), que es ejecutada cuando ninguna de las tareas está en modo **ready**, llamada **OS_TaskIdle()**; ésta siempre estará en la menor prioridad.

10.4.2 Manejo de las secciones críticas de software en uC/OS-II

El manejo de las secciones críticas se realiza usando dos (2) macros para deshabilitar y habilitar las interrupciones del procesador; están presentes en el archivo **OS_CPU.H**, de la siguiente forma:

```
OS_ENTER_CRITICAL(); // Entra a la zona crítica, deshabilita interrupciones
.....
... / Zona Crítica de Software
```

```
OS_EXIT_CRITICAL(); <strong>// Sale de la zona crítica
```

Debe recordarse que el código de la zona crítica debe ser corto, a fin de no tener las interrupciones deshabilitadas por mucho tiempo, lo que podría ocasionar retardos en la atención a los eventos asincrónicos, o peor aún, perderlos.

10.4.3 Portado del uC/OS-II para el AP16A

El µC/OS-II está conformado por 4 secciones:

El código fuente (*source code*), contenido en la página www.micrium.com, es el *kernel* propiamente y está conformado por los siguientes grupos de archivo:

ucos_ii.c

```
#define OS_GLOBALS /* Declare GLOBAL variables */ #include "ucos_ii.h"
#define OS_MASTER_FILE /* Prevent the following files from including includes.h */
#include "os_core.c" //Estructura del kernel: inicialización de listas, eventos, scheduler.
#include "os_flag.c" //Manejo de banderas de bits para las tareas (flags).
#include "os_mbox.c" //Manejo de mensajes entre tareas (mailbox).
#include "os_mem.c" //Manejo de bloques de memoria (buffers).
#include "os_mutex.c" //Manejo de semáforos de exclusión mutua (mutexes).
#include "os_q.c" //Manejo de mensajes en cola (queue).
#include "os_sem.c" //Manejo de semáforos (semaphores).
#include "os_task.c" //Manejo de tareas (tasks): creación, borrado y manejo de stack
//prioridades.
#include "os_time.c" //Manejo de retardos, temporización (timers) y timeouts.
#include "os_tmr.c"
```

Archivos de configuración (µC/OS - II Configuration) **os_cfg_r.h, ucoss_ii.h, includes.h**.

Código de aplicación del usuario (application software)

Código específico del procesador (processor-specific code) **os_cpu.h**

MANEJO DE LAS ZONAS CRÍTICAS PARA EL AP16A:

El archivo posee las 2 funciones para delimitar las zonas críticas, que para la arquitectura HC08 es como sigue:

```
#define OS_ENTER_CRITICAL() {__asm SEI;}
#define OS_EXIT_CRITICAL() {__asm CLI;}
```

Esta definición se realiza en el archivo **OS_CPU.H**. para la configuración, con la definición de **OS_CRITICAL_METHOD** en 1, que es la forma más básica de implementación.

CREACION DEL TICK PARA EL AP16A:

Se debe crear la función de Tick con alguno de los timer del sistema, y además la atención a la interrupción periódica que este genera una vez inicializada, en este caso se usará el timer 1 (canal 0) del AP16A, y la implementación quedará como lo muestra el siguiente código:

```

Timer.C          void          SetCV(unsigned           int          Val){  

    T1CH0H          =          (*(unsigned           char*)&Val);  

    T1MODH          =          (*(unsigned           char*)&Val);  

    T1CH0L          =          (*(((unsigned           char*)&Val)+1));  

    T1MODL          =          (*(((unsigned           char*)&Val)+1));  

}  
  

void          SetPV(unsigned           char          Val){  

    T1SC_PS         =          Val;           /*Almacena      valor      en      prescaler*/  

    T1SC_TRST       =          1;            /*      Reset      al      counter */  

}  
  

void      TimerInit(void){      //Función      de      Inicialización      del      Tick      (10mSeg)  

/*T1SC:                      TOF=0,TOIE=0,TSTOP=1,TRST=1,PS2=0,PS1=0,PS0=0*/  

    T1SC             =          0x30;  

/*T1SC0:CH0F=0,CH0IE=1,MS0B=0,MS0A=1,ELS0B=0,ELS0A=0,TOVO=0,CH0M  

AX=0*/  

    T1SC0           =          0x50;  

    SetCV(0xFFFF);     /*Almacena      el      valor      en      el      registro      de      comparación*/  

    SetPV(0x00);      /*Configura      prescale      de      acuerdo      a      la      velocidad      de      la      CPU */  

/*          T1SC:                      TSTOP=0      */  

    T1SC           &=          ~(byte)(0x20);      /*      Run      counter */  

}  

#pragma          TRAP_PROC  

void      OSTickISR(void){      //Interrupción      periódica      Tick      (10mSeg)  

    T1SC0           &=          ~(byte)(0x80);      //Reset      Interrupt      Flag  

    EnableInterrupts;  

    OSTimeTick();  

    OSIntExit();  

}  

OS_CPU.C  

void          OSStartHardware(void){  

    TimerInit();  

}

```

En el archivo **project.prm** del proyecto se configuran las ISR que serán atendidas por la interrupción Tick llamada **OSTickISR**, y el vector que apunta a la función de cambio de contexto que se invoca al realizar ejecución de la instrucción de interrupción de software **SWI**.

project.prm

```
RAM = READ_WRITE 0x0100 TO 0x045E;  
//modificación de dirección
```

```
STACKTOP 0x045F //modificación del valor inicial //del SP (en  
el tope de la RAM)
```

```
VECTOR 5 OSTickISR //T1CH0: Vector a la función ISR del  
Tick cada 10mSeg VECTOR 1 OSCtxSw //SWI: vector a la función ISR de la interrupción SWI  
VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application.  
*/
```



Puede adaptar el sistema uC/OS-II a cualquier arquitectura de microcontrolador con manejo de stack y que permita que los registros internos puedan almacenarse ante una interrupción.

EJEMPLO No. 36

Manejo de RTOS uC/OS-II

Objetivo:

Realizar todas las configuraciones necesarias que faciliten trabajar cualquier proyecto en C con el microcontrolador MC68HC908AP16A de Freescale™ usando el sistema operativo uC/OS-II. Hacer un proyecto ejemplo que realice la prueba de la configuración, escribir un programa en C usando el sistema operativo que maneje 2 tareas, Tarea1 maneja la temporización del Led1 (OUT-1) 500mSeg encendido y 500mSeg apagado, Tarea2 maneja la temporización del Led2 (OUT-2) 1 Segundo encendido y 1 Segundo apagado.

Solución:

El proyecto en Codewarrior® tendrá el siguiente aspecto:

Donde los archivos

```

void ToggleLed
    DDRC_DDRC2 =
    PTC_PTC2 ^=
}

void Task1(voi
    pdata = pdat
    OSStartHardw
    handlerTas
    for(;;){
        ToggleLe
        OSTimeD1
    }
}

void Task2(voi
    pdata = pd
    for(;;){
        ToggleLe
        OSTimeD1
    }
}

```

OS_CPU_A.ASM: contiene código en lenguaje de ensamblador para las funciones críticas del sistema operativo como son las de cambio de contexto (**OSCtxSw**), salida de interrupción (**OSIntExit**). Para la configuración del **AP16A** solo se requiere que este corresponda al código del HC08 con esto podría trabajar para cualquier miembro de los procesadores de 8 bits de arquitectura HC08, incluyendo Flexis™ de 8 bits. Por lo tanto este archivo no requiere configuración alguna.

TIMER.C: Contiene el cuerpo de la función Timer_Init() que se encarga de la generación del Tick que en este caso se ha seleccionado de 10 mili Segundos, además de la ISR del Tick, el código básico de este

módulo es:

```
void SetCV(unsigned int Val){  
    T1CH0H = (*(unsigned char*)&Val);  
    T1MODH = (*(unsigned char*)&Val);  
    T1CH0L = (*(((unsigned char*)&Val)+1));  
    T1MODL = (*(((unsigned char*)&Val)+1));  
}  
  
void SetPV(unsigned char Val){  
    T1SC_PS = Val; /*Almacena valor en prescaler*/  
    T1SC_TRST = 1; /* Reset al counter */  
}  
  
void TimerInit(void){ //Función de Inicialización del Tick  
/*T1SC:TOF=0,TOIE=0,TSTOP=1,TRST=1,PS2=0,PS1=0,PS0=0*/  
    T1SC = 0x30;  
/*T1SC0:CH0F=0,CH0IE=1,MS0B=0,MS0A=1,ELS0B=0,ELS0A=0,TOVO=0,CH0MAX=0*/  
    T1SC0 = 0x50;  
    SetCV(0x5FFF); /*Almacena el valor apropiado en el registro de comparación*/  
    SetPV(0x00); /*Configura prescaler*/  
/* T1SC: TSTOP=0 */  
    T1SC &= ~(byte)(0x20); /* Run counter */  
}  
  
#pragma TRAP_PROC  
  
void OSTickISR(void){ //Interrupción periódica Tick
```

```

T1SC0 &= ~(byte)(0x80); //Reset Interrupt Flag

EnableInterrupts;

OSTimeTick();

OSIntExit();

}

```

OS_CPU_C.C: Se adiciona al proyecto tal cual como la trae el uC/OS-II, solo una función es alterada OSStartHardware(), para que invoque la función de inicialización del Tick TimerInit().

```

void OSStartHardware(void){

    TimerInit();

}

```

OS_CPU.H: Configurar los macros para entrar y salir de las zonas críticas y el macro de cambio de contexto usando la instrucción SWI que genera la interrupción respectiva del Software:

```

#define OS_CRITICAL_METHOD 1

#if OS_CRITICAL_METHOD == 1

#define OS_ENTER_CRITICAL() {__asm SEI;}

#define OS_EXIT_CRITICAL() {__asm CLI;}

#endif

#define OS_TASK_SW() {__asm SWI;}

```

Project.PRM: En el archivo de configuración de direcciones de las interrupciones usadas por el RTOS y ampliar el valor del stack pointer al tope de la RAM, que para el caso sería:

```
RAM = READ_WRITE 0x0100 TO 0x045E; /*La RAM de usuario se disminuye en 1*/
```

```
//STACKSIZE 0x50
```

```
STACKTOP 0x045F /*SP inicia en el tope de la RAM*/
```

```
VECTOR 5 OSTickISR /*Vector de la interrupción del Timer*/
```

```
VECTOR 1 OSCtxSw /*Vector de la interrupción SWI*/
```

```
VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application. */
```

Finalmente, el código del programa principal que resuelve el enunciado es:

```
/****** Ejemplo 36 *****
```

```
// Manejo de proyectos con RTOS uC/OS-II
```

```
// Fecha: Abril 16,2009
```

```
// Asunto: Configuración y uso del
```

```
// Sistema Operativo uC/OS-II
```

```
// para Sistemas Embebidos
```

```
// Hardware: Sistema de desarrollo AP-Link
```

```
// Freescale™.
```

```
// Versión: 1.0 Por: Gustavo A. Galeano A.
```

```

//*****  

#include <hdef.h>  

#include “derivative.h”  

#include “ucos_ii.h”  

#define TASK_STK_SIZE 40 //Tamaño de stack de Tareas  

INT8U Task1Stk[TASK_STK_SIZE]; //Stack Tarea 1  

INT8U Task2Stk[TASK_STK_SIZE]; //Stack Tarea 2  

INT8U handlerTask1,handlerTask2; //Handles  

void Task1(void *pdata);  

void Task2(void *pdata);  

void ToggleLed1(void){//Invierte estado del Led ON/OFF  

    DDRC_DDRC3 = 1;  

    PTC_PTC3 ^= 1;  

}  

void ToggleLed2(void){//Invierte estado del Led ON/OFF  

    DDRC_DDRC2 = 1;  

    PTC_PTC2 ^= 1;  

}  

void Task1(void* pdata){//Tarea 1  

    pdata = pdata;  

    OSStartHardware();  

    handlerTask2 = OSTaskCreate(Task2, (void*)0, (void*)&Task2Stk[TASK_STK_SIZE],9);  

    for(;;){  

        ToggleLed1();
}

```

```

OSTimeDly(50);/* Retardo de 0.5 Segundo*/
}

}

void Task2(void* pdata){//Tarea 2

pdata = pdata;

for(;;){

    ToggleLed2();

    OSTimeDly(100); /* Retardo de 1 Segundo*/

}

}

void Mcu_Init(void){ //Inicialización del MCU

CONFIG1 = 0x03; //STOP=1 COPD=1

CONFIG2 = 0x00;

}

void main(void){//Función principal

Mcu_Init();

OSInit();

handlerTask1 = OSTaskCreate(Task1,(void*)0,(void*)&Task1Stk[TASK_STK_SIZE], 8);

OSStart();

}

```

Discusión:

Para la configuración del RTOS del HC08AP16A, se crea el proyecto en Codewarrior® de la

forma habitual, se crean a su vez 4 grupos de archivos llamados Ports, Application, Configuration y Sources_OS a esta última se le adicionan los archivos fuente del uC/OS-II ucous_ii.c y ucos_ii.h, los cuales incluyen a sus vez todos los archivos requeridos por el sistema: os_core.c, os_flag.c, os_mbox.c, os_mem.c, os_mutex.c, os_q.c, os_sem.c, os_task.c, os_time.c y os_tmr.c, archivos que deberán estar en la carpeta del archivo uc_os_ii.c.

En la carpeta *Ports* se agregan al proyecto los archivos **OS_CPU_A.ASM**, **OS_CPU_C.C** y **Timer.C** (este con la modificación descrita en la solución).

Los vectores asociados a las interrupciones usadas por el RTOS para el *Tick* y el cambio de contexto son configurados en el archivo **Project.PRM** en la carpeta Project *Settings\Linker Files*. Finalmente, en la carpeta *Application* el archivo main.c contiene la aplicación final que usa el sistema.



Manejo de RTOS
uC/OS-II: proyecto completo para Freescale™ (AP-Link) y Flexis™.

De esta forma el proyecto queda configurado para soportar cualquier proyecto con el microcontrolador **AP16A** y con cambios en **Timer.C** para la generación

del Tick podrá ser portado a cualquier **HC08** o **Flexis™** de 8 bits.

RESUMEN DEL CAPÍTULO

Un sistema operativo de tiempo real o RTOS, es un programa que coordina la operación de otros programas llamados tareas. El uso de estos sistemas en un proyecto embebido proporciona gran cantidad de ventajas, especialmente en un equipo de trabajo (R&D);

sin embargo, se debe recordar que al mismo tiempo es un programa que ocupa recursos, espacio en la memoria del microcontrolador y toma tiempo para realizar la labores de coordinación.

Debido a lo anterior, no en todos los proyectos embebidos es recomendable usar un RTOS como plataforma de programación, en especial cuando se tienen recursos de velocidad y memoria limitadas, como es el caso de los procesadores de 8 bits; sin embargo, existen técnicas (foreground/background) que aunque no son RTOS en todo el sentido de la palabra, involucran conceptos de ellos que pueden resolver el problema de la simultaneidad y la continuidad de una forma aceptable para la aplicación.

La tarea que ejecuta el RTOS será siempre la de mayor prioridad que se encuentre en estado **ready**, de esta forma un **kernel** preemptivo, garantiza que el tiempo de respuesta a las tareas más importantes es corto, lo que sumado a una frecuencia media-alta del reloj (**tick**), da la sensación de “tiempo real” en todas las tareas activas, cuando en realidad solo se está ejecutando una tarea en un determinado tiempo.

El kernel no-preemptivo proporciona un esquema más sencillo de implementar, toma menos recursos de la máquina y contiene menos preámbulo (overhead); sin embargo, y a diferencia de los preemptivos, los tiempo de respuesta de las tareas prioritarias son más largos, de modo que es la aplicación final la que sugiere si un sistema usa uno u otro con mayor eficiencia, o si por simplicidad el sistema *foreground/background* es suficiente.

PREGUNTAS Y EJERCICIOS PROPUESTOS

Qué desventajas tiene el uso de un RTOS en un microcontrolador de 8 bits?

Qué características serían relevantes al seleccionar un microcontrolador sobre el cual se montará un RTOS?

Mencione dos ventajas y dos desventajas de cada uno de los kernels: no-preemptivo y preemptivo. En un RTOS de kernel preemptivo, ¿puede la tarea que está en estado run ser suspendida, siesta cambia la prioridad de otra tarea a un nivel mayor? Justifique la respuesta.

Cómo soluciona un RTOS el hecho de que varias tareas intenten accesar un mismo recurso en un tiempo determinado?

¿Podrá combinarse el uso de herramientas como el Processor Expert™ con el uso de un RTOS para la capa de aplicación?

¿Qué problema podría tener un RTOS que permita interrupciones anidadas? ¿Cómo minimizar el problema?

Compare las ventajas y desventajas de la siguiente situación: usar un sistema preemptivo a una frecuencia de Tick x vs. usar un sistema No preemptivo de frecuencia de tick 2x. ¿Bajo qué condiciones sería mejor usar uno u otro? Analice velocidad, tiempo de respuesta, consumo de energía, facilidad, robustez.

TABLA ASCII

DEC HEX CHAR	DEC HEX CHAR	DEC HEX CHAR	DEC HEX CHAR
0 0x00 NULL	32 0x20 Espacio	64 0x40 @	96 0x60 `
1 0x01 SOH	33 0x21 !	65 0x41 A	97 0x61 a
2 0x02 STX	34 0x22 ``	66 0x42 B	98 0x62 b
3 0x03 ETX	35 0x23 #	67 0x43 C	99 0x63 c
4 0x04 EOT	36 0x24 \$	68 0x44 D	100 0x64 d
5 0x05 ENQ	37 0x25 %	69 0x45 E	101 0x65 e
6 0x06 ACK	38 0x26 &	70 0x46 F	102 0x66 f
7 0x07 BEL	39 0x27 `	71 0x47 G	103 0x67 g
8 0x08 BSpac	40 0x28 (72 0x48 H	104 0x68 h
9 0x09 TAB	41 0x29)	73 0x49 I	105 0x69 i
10 0x0A LFeed	42 0x2A *	74 0x4A J	106 0x6A j
11 0x0B VTab	43 0x2B +	75 0x4B K	107 0x6B k
12 0x0C FF	44 0x2C ,	76 0x4C L	108 0x6C l
13 0x0D CR	45 0x2D -	77 0x4D M	109 0x6D m
14 0x0E SO	46 0x2E .	78 0x4E N	110 0x6E n
15 0x0F SI	47 0x2F /	79 0x4F O	111 0x6F o
16 0x10 DLE	48 0x30 0	80 0x50 P	112 0x70 p
17 0x11 DC1	49 0x31 1	81 0x51 Q	113 0x71 q

18 0x12 DC2	50 0x32 2	82 0x52 R	114 0x72 r
19 0x13 DC3	51 0x33 3	83 0x53 S	115 0x73 s
20 0x14 DC4	52 0x34 4	84 0x54 T	116 0x74 t
21 0x15 NAK	53 0x35 5	85 0x55 U	117 0x75 u
22 0x16 SYN	54 0x36 6	86 0x56 V	118 0x76 v
23 0x17 ETB	55 0x37 7	87 0x57 W	119 0x77 w
24 0x18 CAN	56 0x38 8	88 0x58 X	120 0x78 x
25 0x19 EM	57 0x39 9	89 0x59 Y	121 0x79 y
26 0x1A SUB	58 0x3A :	90 0x5A Z	122 0x7A z
27 0x1B ESC	59 0x3B ;	91 0x5B [123 0x7B {
28 0x1C FS	60 0x3C <	92 0x5C \	124 0x7C
29 0x1D GS	61 0x3D =	93 0x5D]	125 0x7D }
30 0x1E RS	62 0x3E >	94 0x5E ^	126 0x7E ~
31 0x1F US	63 0x3F ?	95 0x5F _	127 0x7F □

LIBRERÍAS DE FUNCIONES ESTÁNDAR DEL C

Librería <math.h>

double	acos	(double	x);
float	acosf	(float	x);
double		asin(double	x);
float		asinf(float	x);
double	atan	(double	x);
float		atanf(float	x);
double	atan2(double	y,	double
			x);

```

float atan2f(float y, float x);
double ceil(double x);
float ceilf(float x);
double cos((double) x);
float cosf(float x);
double coshf(float x);
float fabs((double) x);
double fabsf(float x);
float floor((double) x);
double floorf(float x);
double fmod((double) x, double y);
float fmodf(float x, float y);
double frexp(double x, int *exp);
float frexpf(float x, int *exp);
double ldexp((double) x, int exp);
float ldexpf(float x, int exp);
double log((double) x);
float logf(float x);
double log10(double x);
float log10f(float x);
double modf(double x, double *i);
float modff(float x, float *i);
double pow((double) x, double y);
float powf(float x, float y);
double sin(double x);
float sinf(float x);
double sinh(double x);
float sinhf(float x);
double sqrt(double x);
float sqrtf(float x);
double tan(double x);
float tanf(float x);
double tanh(double x);
float tanhf(float x);

```

Librería <stdlib.h>

```

int abs(int i);
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
void *bsearch(const void *key,const void *array,size_t n,size_t size, cmp_func cmp());
void *calloc(size_t n, size_t size);
div_t div(int x, int y);
void free(void *ptr);
long labs(long i);
ldiv_t ldiv(long x, long y);

```

```

void *malloc(size_t size);
int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *wcs, const char *mbs, size_t n);
int mbtowc(wchar_t *wc, const char *s, size_t n);
void *qsort(const void *array, size_t n, size_t size, cmp_func cmp);
int rand(void);
void *realloc(void *ptr, size_t size);
void srand(unsigned int seed);
double strtod(const char *s, char **end);
long strtol(const char *s, char **end, int base);
unsigned long strtoul(const char *s, char **end, int base);

```

Librería <stdio.h>

```

int getchar(void);
char *gets(char *s);
int printf(const char *format, ...);
int putc(char ch, FILE *f);
int putchar(char ch);
int puts(const char *s);
int scanf(const char *format, ...);
int sprintf(char *s, const char *format, ...);

```

Librería <string.h>

```

void *memchr(const void *p, int ch, size_t n);
void *memcmp(const void *p, const void *q, size_t n);
void *memcpy(const void *p, const void *q, size_t n);
void *memmove(const void *p, const void *q, size_t n);
void *memset(void *p, int val, size_t n);
time_t mktime(struct tm *time);
char *strcat(char *p, const char *q);
char *strchr(const char *p, int ch);
int strcmp(const char *p, const char *q);
char *strcpy(char *p, const char *q);
size_t strcspn(const char *p, const char *q);
char *strerror(int errno);
size_t strftime(char *s, size_t max, const char *format, const struct tm *time);
size_t strlen(const char *s);
char *strncmp(char *p, const char *q, size_t n);
char *strncpy(char *p, const char *q, size_t n);
char *strpbrk(const char *p, const char *q);
char *strrchr(const char *p, int c);
size_t strspn(const char *p, const char *q);
char *strstr(const char *p, const char *q);
char *strtok(char *p, const char *q);

```

Librería <ctype.h>

```

int isalnum
int isalpha
int iscntrl(int ch);
int isdigit(int ch);
int isgraph(int ch);
int islower(int ch);
int isprint(int ch);
int ispunct(int ch);
int isspace(int ch);
int isupper(int ch);
int isxdigit(int ch);
int tolower(int ch);
int toupper(int ch);

```

Librería <time.h>

char *	asctime(const	struct	tm*	timeptr);
clock_t				clock(void);
char	*ctime(const	time_t		*timer);
double	difftime(time_t		time_t	t0);
struct	tm	*t1,	time_t	*time);
struct	tm	*gmtime(const	time_t	*time);
		*localtime(const	time_t	
time_t	time(time_t *timer);			

GLOSARIO

Acelerómetro: dispositivo semiconductor que provee señales externas que indican el nivel de aceleración física en un valor analógico.

ADC (Analog to Digital Converter): módulo que convierte el valor de una señal analógica del exterior a una representación binaria.

ALU (Aritmetical Logic Unit): unidad interna de la CPU encargada del procesamiento aritmético.

Array: arreglo, conjunto de datos consecutivos en memoria de un mismo tipo.

ASCII (American Standard Code for Information Interchange): código estándar de identificación de caracteres usado en programación para manipular caracteres que se pueden imprimir.

Bean: componente encapsulado de software funcional, el cual puede ser incluido en un proyecto y parametrizado.

Bluetooth: tecnología inalámbrica usada para transmisión de voz a corta distancia, frecuentemente utilizada como sistema de manos libre en equipos celulares.

Breakpoint: en español punto de ruptura; es un punto dentro del programa que señala el lugar en el cual se requiere que el programa sea suspendido para verificar el estado de registros, memoria y periféricos. útil en la depuración de aplicaciones embebidas.

Buffer: sección de RAM usada para almacenamiento de datos de forma temporal.

Bug: error en el funcionamiento de un software.

Cambio de contexto (Context Switching): proceso de cambio de ejecución del procesador de una tarea a otra.

Carácter: dato que representa un símbolo de un byte.

Cast: molde, ajuste que provee al compilador una forma de operar y de almacenar variables de diferente tipo.

CCR (Condition Code Register): registro interno de la CPU en procesadores Freescale de 8 bits, que indica en bits el estado de una operación aritmética o lógica.

Char: tipo de datos de un byte. Mínima declaración de variable en C.

Ciclo útil: para una onda digital, representa el porcentaje de tiempo que la señal esta activa.

CISC (Complex Instruction Set Computer): tipo de arquitectura interna de procesadores que está basado en instrucciones que realizan procesos complejos en una sola ejecución.

CLI (Clear Interrupt Flag): instrucción de procesadores Freescale que permite habilitar el suceso de interrupciones.

Compilador: programa que toma un código en texto plano y lo convierte a instrucciones de máquina de otro procesador.

Const: modificador usado para indicar que el lugar de almacenamiento de una variable se realiza en la memoria FLASH, en lugar de hacerlo en la memoria RAM.

Constante: tipo de variable en C que tiene un valor inicial y no puede ser cambiado; normalmente está en memoria de programa (flash), puede ser leída, mas no escrita.

COP (Computer Operating Properly): módulo interno de protección de bloqueo por código redundante usado en microcontroladores.

CPU (Central Process Unit): unidad de procesamiento central de un microcontrolador, centro de operación y decodificación de instrucciones.

Depurador: parte del IDE que permite realizar seguimiento a un programa enviado al microcontrolador. Permite ubicar secciones del programa que no funcionan de acuerdo a lo especificado.

Desktop: computador de mesa, utilizado en aplicaciones no embebidas, como equipo de oficina o servidor.

Driver: módulo de software que permite el manejo de algún periférico externo. Contiene funciones precisas de acceso y manejo.

Double: tipo de datos en C de cuatro bytes de longitud, posibilita representar variables cuyos rangos de valores son extensos.

EPRO M (Electrical Erasable Programable Read Only Memory): tipo de memoria de almacenamiento que no pierde su contenido al ser desenergizada, borrable y programable por medio eléctrico.

Encapsulado: código de programa ya ensayado y que tiene solo ciertos procedimientos de acceso.

Ensamblador: subprograma que convierte un código en lenguaje en C a su correspondiente código en lenguaje de ensamblador.

Evento: suceso de notificación a una tarea que indica la ocurrencia de un hecho o estímulo.

Extern: modificador en C que indica que la declaración de una variable se encuentra en un módulo o archivo externo al que se está usando.

FAA (Federal Aviation Administration): organismo internacional encargado de la seguridad de la aviación civil.

Far: lejano, modificador del C que indica que la variable debe ser almacenada en la zona de acceso extendido de la RAM.

FFT (Fast Fourier Transform): algoritmo de conversión de señales analógicas en el tiempo a su representación en componentes senoidales (dominio de la frecuencia).

FIR (Finite Impulse Response): tipo de filtro digital que se basa en la respuesta del sistema a una señal de tipo impulso.

Flash: tipo de memoria no volatil que se borra y programa eléctricamente; es un tipo de EEPROM de menor precio que puede ser borrada en grandes bloques.

Float: flotante, tipo de almacenamiento que representa un valor real con una precisión específica.

FOB (Free on Board): es el precio de venta de los bienes embarcados a otros países, puestos en el medio de transporte sin incluir valor de los seguros y fletes.

Foreground/Background: consecutivo/interrupción, técnica de programación que basa su funcionamiento en el llamado consecutivo de tareas, donde los eventos asincrónicos son atendidos por las interrupciones.

Full Chip Simulation: técnica de depuración que no requiere hardware conectado para la evaluación; el funcionamiento está 100% simulado en un desktop.

Full Duplex: técnica de comunicación entre dos dispositivos en la cual los datos pueden ser transportados en ambos sentidos al mismo tiempo. Un equipo full-duplex puede estar enviando un dato, mientras está recibiendo otro en el mismo instante.

GPS (Global Positioning System): sistema de posicionamiento global usado para propósitos de navegación. Utiliza satélites de media órbita para determinar la posición en la que se encuentra un equipo móvil que contiene un receptor de señal GPS.

GUI (Graphical User Interface): es un tipo de interfaz que permite al humano interactuar con un equipo electrónico por medio de controles gráficos.

HAL (Hardware Abstraction Layer): capa de programación superior en la que el usuario interactúa con un hardware sin que se requiera conocer los detalles que lo componen.

I2C (Inter-Integrated Circuit): estándar de comunicación serial sincrónica entre periféricos, que usa dos líneas SDA y SCL.

ICG (Internal Clock Generator): módulo generador de señal de reloj interno, que en muchas aplicaciones puede reemplazar el oscilador externo basado en cristales de cuarzo.

IDE (Integrated Development Environment): ambiente de desarrollo en software que incorpora los componentes necesarios (editor, compilador, programador, depurador) para desarrollar un proyecto.

Int: entero, tipo de almacenamiento de 16 bits usado en variables.

Interpretador: programa residente en un procesador que toma, línea a línea, instrucciones de alto nivel y las convierte en lenguaje de máquina en tiempo de ejecución.

Interrupción: evento asincrónico que suspende la ejecución normal de un programa por un corto período de tiempo.

IRQ: pin de entrada de microcontroladores que genera interrupción a la CPU cuando en ella se genera un cambio programado, ya sea que la señal cambie su estado digital de uno (1) a cero (0), o de cero (0) a uno (1).

ISR (Interrupt Service Routine): función invocada cuando se presenta una interrupción a la CPU.

KBI (Keyboard Interrupt): módulo interno de entrada de microcontroladores, especialmente usado en la lectura de teclas externas.

Kernel: componente fundamental de un sistema operativo, encargado de distribuir el tiempo de la CPU para el manejo de las tareas.

Latencia de interrupción: tiempo que transcurre desde que una interrupción sucede y la ejecución de la primera instrucción de la función respectiva de interrupción (ISR).

Librería: grupo de funciones estándar del ANSI C que pueden ser accesados desde una aplicación para realizar una operación específica con datos entregados.

Linker: enlazador, último subprograma del compilador que convierte el código objeto (.OBJ) en instrucciones propias del procesador final (target) en direcciones fijas del mapa de memoria.
Long: entero doble, tipo de almacenamiento de variables de 4 bytes.

LVI (Low Voltage Inhibit): módulo interno sensor del nivel de voltaje de alimentación del microcontrolador, permite inhibir la operación de la CPU si el valor de voltaje está por debajo de un valor permitido.

Mapa de memoria: distribución de los bloques de almacenamiento de memoria y registros que contiene un procesador.

MISO (Master Input Slave Output): pin estándar de la comunicación SPI que identifica la entrada de datos al maestro, provenientes del dispositivo esclavo.

MOSI (Master Output Slave Input): pin estándar de la comunicación SPI que identifica la salida de datos del maestro hacia el dispositivo esclavo.

MP3: MPEG-1 Audio layer 3, formato de codificación digital de audio usando algoritmos de compresión que permiten ahorrar espacio de almacenamiento.

Near: modificador de variable en C que indica que a la variable debe asignársele una dirección dentro de la zona directa (0x00 a 0xFF), normalmente usado en variables de acceso rápido.

PC (Program Counter): registro interno de la CPU que contiene la dirección de la instrucción que se está ejecutando en determinado momento.

Pointer: apuntador, objeto que contiene la dirección de un dato o variable. Usado como argumento en el paso a funciones por referencia.

Polling: técnica de programación que se basa en el muestreo frecuente de las entradas para identificar un evento.

POO: (OOP: Object Oriented Programming): programación orientada a objetos, se refiere a una técnica de programación que utiliza nivel de abstracción que consiste en dividir un problema en piezas manejables.

Portabilidad: término usado para indicar que un código de programa es reutilizable y que puede ser pasado a otra arquitectura con pocos o ningún cambio.

PLL (Phase Locked Loop): módulo interno de ciertos procesadores que utiliza la técnica de enganche de fase para generar una frecuencia estable de oscilador que estimula la CPU.

Prototipo: forma de una función, indica en una línea el tipo de argumentos que recibe y el retorno final resultante del llamado a una función.

Pre-procesador: primer subprograma del compilador que se encarga de la revisión semántica y solución a constantes.

Prioridad de ejecución: atributo de una tarea que indica el grado comparativo de importancia de ejecución con respecto a otras.

Processor Expert™: componente de software del Codewarrior IDE que permite generar código de inicialización de periféricos de un microcontrolador.

R&D (Research and Development): investigación y desarrollo, grupo de diseñadores que desarrollan productos basados en una especificación externa.

RAD (Rapid Application Design): metodología de desarrollo de software que involucra herramientas que facilitan la construcción de prototipos de forma rápida. **Recurso:** entidad (software o hardware) usado por una tarea con un fin específico.

Recursividad: llamado de una función a sí misma. **Registe:** modificador de C usado para indicar que el almacenamiento de una variable debe realizarse en uno de los registros internos del procesador.

Reset: señal de entrada de un microcontrolador que indica suspender el proceso actual y reiniciar su labor desde el punto inicial (Startup).

RISC (Reduced Instruction Set Computer): tipo de tecnología de arquitectura interna que se basa en la elaboración de pocas instrucciones rápidas y eficientes para su procesamiento.

Round-Robin: técnica de programación que se basa en el llamado consecutivo de procedimientos.

RTI / RETI (Return From Interrupt): instrucción de ciertos procesadores para retornar de la ISR.

RTOS (Real Time Operating System): programa residente que controla el funcionamiento de varias tareas, dando a cada una la sensación de simultaneidad.

RS232: especificación de bajo costo estándar de transmisión serial asincrónica en dos hilos.

RTC (Real Time Clock): módulo de hardware y software específico para el manejo del tiempo en formato calendario.

Scheduler: parte del kernel que se encarga de determinar la tarea con mayor prioridad a la cual se le debe entregar el control de la CPU.

SCL: línea de sincronización usada en comunicaciones I2C.

SDA : línea de datos bidireccional usada en comunicaciones I2C.

SEI (Set Interrupt Flag): instrucción usada para deshabilitar las interrupciones antes de entrar a una zona crítica de software.

Semáforo: mecanismo de control de los RTOS para el manejo ordenado de los recursos de un sistema.

SFR (Special Function Registers): registro de configuración en microcontroladores PIC.

Signed: modificador de variables que indica que la variable a almacenar puede contener valores tanto positivos como negativos. Establece que el bit de mayor peso de una variable indica el signo.

Single-Chip: pastilla única de procesamiento que contiene internamente todo lo necesario para realizar procesamiento.

Sistema operativo: programa residente que controla el funcionamiento de varias tareas, dando a cada una la sensación de simultaneidad.

Sizeof: operador tiempo de compilación que entrega la medida en bytes usados por una variable o una expresión.

SLEP: modo de bajo consumo de procesadores, que suspende el procesamiento interno.

SP (Stack Pointer): apuntador de pila, registro que apunta a una dirección del stack disponible para almacenamiento de datos.

SS (Slave Select): señal de selección de dispositivo esclavo por parte del maestro usado en comunicaciones SPI.

Stack: pila, zona de RAM de los procesadores, dedicada al almacenamiento de datos temporales, información de registros y direcciones de retorno.

Stack Pointer: apuntador de pila, registro que apunta a una dirección del stack disponible para almacenamiento de datos.

Stacking: proceso automático de almacenamiento de registros internos del procesador al suceder en la CPU una interrupción.

Startup Code: pequeño código de inicialización que se ejecuta previo al llamado de la función main(). Inicializa stack pointer y variables.

STOP: instrucción y modo de muy bajo consumo de los procesadores, que permite reducir la corriente a niveles inferiores.

SWI (Software Interrupt): instrucción que genera secuencia de interrupción una vez ejecutada. Permite direccionar el programa a una ISR definida.

Task: tarea, procedimiento formado por varias líneas de código que realizan una labor específica.

TBM (Time Base Module): módulo interno de ciertos microcontroladores que se encargan de generar una señal muy precisa para contabilizar tiempo.

Tick: interrupción de tiempo periódica usada en los sistemas operativos para el manejo del tiempo real y el cambio de tarea por parte del scheduler.

Tiempo de compilación: tiempo que se tarda una computadora en convertir un código en lenguaje de alto nivel (con el C), al correspondiente lenguaje de máquina.

Tiempo de ejecución: tiempo que se tardan las instrucciones en el microcontrolador en ejecutar un código.

Timeout: tiempo de espera máximo para la ocurrencia de un evento.

Timer Overflow: sobreflujo del temporizador, paso del valor máximo al valor mínimo de un contador.

Tfall: tiempo que tarda una señal digital en pasar de su estado lógico uno (1) a su estado lógico cero (0). Este tiempo en teoría es nulo pero en la práctica no.

Transientes: evento momentáneo indeseable en sistemas de energía. Cambio en una variable que desaparece durante una transición desde un estado a otro.

Trise: tiempo que tarda una señal digital en pasar de su estado lógico cero (0) a su estado lógico uno (1). Este tiempo en teoría es nulo pero en la práctica no.

TTL (Transistor Transistor Logic): es un tipo de fabricación de lógica digital.

TTM (Time To Market): tiempo transcurrido entre la concepción del diseño de un producto y su inicio de producción.

Unsigned: modificador usado para indicar que una variable solo tomará valores positivos.

USART (Universal Asynchronous Receiver Transmitter): estándar de intercambio de datos que permite transmisión y recepción de datos de forma full-duplex a diferentes velocidades de configuración.

Variable: localizaciones en memoria para el almacenamiento de un dato.

Volatile: modificador en C que indica que el contenido de una variable puede cambiar de forma no determinada por el compilador, típicamente por efecto de una interrupción.

WAIT: modo de bajo consumo de los procesadores, que permite que algunos periféricos internos continúen su ejecución.

Warning: mensaje de precaución emitido por el compilador cuando existe una posible fuente de error en tiempo de ejecución.

WAV (Waveform Audio Format): es un formato estándar de almacenamiento de audio no comprimido.

Zona crítica: sección de software que una vez iniciado es ejecutado sin permitir interrupciones.

BIBLIOGRAFÍA

Bannoura, Munir. Bettelheim, Rudan. And Soja, Richard. COLDFIRE® Microprocessors & Microcontrollers,

Arquitecture & Programming. AMT Publishing, Farmington Hills, Michigan, 2006. ISBN 0-9762973-0-2.

Benson, David, C What Happens, Using Microcontrollers and The CCS C Compiler, Version 1.0, Square 1 Electronics P.O. Box 1414, Hayden ID 83835.

Freescale Semiconductor, MC68HC908AP64A Data Sheet, M68HC08 Microcontrollers, Rev. 3 10/2007.

Karim Nice, How Stuff Works “How Anti-lock Brakes Work” <http://www.howstuffworks.com/anti-lockbrake.htm>

Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Second Edition. Englewood Cliffs, N.J.: Prentice Hall, 1978.

Labrose, Jean J. Embedded Control Systems Building Blocks. USA: Lawrence, KS, 1995.

Labrose, Jean J. MicroC/OS-II The Real Time Kernel Second Edition, CMPBooks San Francisco, CA 2002.

Motorola, Application Note AN2616 2/2004 Getting Started with HCS08 and CodeWarrior Using C (Stephen Pickering), East Kilbride, Scotland.

Microchip, DS39582B: PIC16F87XA Data Sheet, 28/40/44-Pin, Enhanced Flash Microcontrollers 2003 Microchip Technology Inc.

Motorola, CPU08 Central Processor Unit, Reference Manual, 08/96.

Pereira, Fabio, HCS08 Unleashed, Designer’s Guide to the HCS08 Microcontrollers, Booksurge 2008

Schildt, Herbert. Turbo C/C++: Manual de Referencia. México: Mc Graw Hill, 1992. Stracker, David. C-Style, Standards and Guidelines, Prentice Hall, 1991.

Stracker, David C-Style, Standards and Guidelines, Prentice Hall, ISBN 0-13-116898-3.

Valvano, Jonathan W. Developing Embedded Software in C using ICC11/ICC12/Metrowerks. Texas University: Thomson-Engineering Publishers, ISBN 0-534-39177-x.
<http://users.ece.utexas.edu/~valvano/embed/toc1.htm>-<http://www.freeiconsweb.com/DelliPack-icons.html>

Valvano, Jonathan W. Embedded Microcomputer Systems. University of Texas. Brooks/Cole (Thomson Learning). 2000. ISBN 0 534-36642-2.

Van Sickle, Ted., Programming Microcontrollers in C, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

Otros materiales consultados:

24AA04/24LC04B 4K I2C Serial EEPROM Memory. Microchip Technology 2009. AE109: Processor Expert Tips and Tricks. Freescale Technology Forum 2008. Ruth Rhoades / Petr Struzka.

AE112: Turning Legacy Code into a Reusable Library (using Processor Expert Bean Wizard). Freescale Technology Forum FTF 2008. Ruth Rhoades / Petr Struzka.

AN3467 Rev. 0,05/2007 Freescale™ Semiconductor- Application Note: Using Processor Expert with Flexis™ Microcontrollers.

CMOS Logic Databook, National Semiconductor 1988.
Freescale MCF51QE128 Reference Manual, Rev 3 09/2007.
Archivo de ayudas del software Codewarrior de Freescale.
Hojas de datos acelerómetro MMA7260Q www.freescale.com

Páginas Web:

www.ccsinfo.com
www.faa.gov
www.freescale.com/BeanStore
www.micrium.com
www.processorexpert.com
<http://www.sudokukingdom.com/daily-sudoku-puzzle.php>
www.uCOS-II.com
www.microchip.com
www.iso.org