
TERCER CURSO. INFORMÁTICA INDUSTRIAL I I

Escuela Politécnica Superior
Universidad de Huelva

# Tema 1: Programación de Microcontroladores en C

Manuel Sánchez Raya
Versión 1.0
28 de Enero de 2004

# ÍNDICE

1.- Introducción. ..... 3
1.1.- ¿Que es un sistema empotrado? ..... 3
1.2.- Variaciones del tema ..... 5
1.3.- C: el común denominador ..... 7
1.4.- Unas palabras acerca el Hardware ..... 8

2.- Primer programa en C. ..... 9
2.1.- Hola Mundo! ..... 9
2.2.- .El LED parpadeante ..... 10
2.2.1.- toggleLed. ..... 11
2.2.2.- delay ..... 12
2.3.- El papel del bucle infinito ..... 13
2.4.- Documentación en el desarrollo de software. ..... 13
2.5.- Anatomía de un programa. ..... 20
2.5.1.- Atención a sensores. ..... 20
2.5.2.- Interrupciones. ..... 22

3.- El proceso de Compilar, Linkar y Cargar. ..... 24
3.1.- El proceso de producción del software. ..... 24
3.2.- Compilación ..... 25
3.3.- Linkado ..... 27
3.4.- Ubicación en Memoria del Programa. ..... 29

4.- Descarga y depuración. ..... 31
4.1.- Cuando lo hacemos en ROM… ..... 31
4.3.- Depuradores remotos o monitores. ..... 32
4.4.- Emuladores en circuito. ..... 33
4.5.- Simuladores y otras herramientas. ..... 35

5.- Metodología para conocer el hardware. ..... 36
5.1.- Análisis de Sistemas. ..... 36
5.1.1.- Estudio del esquema eléctrico. ..... 36
5.1.2.- Diagrama de Bloques. ..... 37
5.2.- Diseño de sistemas. ..... 38
5.2.- Mapa de Memoria. ..... 39
5.3.- Inicialización de Software y Hardware. ..... 40
5.4.- Tipos de Memoria del Sistema. ..... 41
5.5.- Técnicas de testeo de memoria. ..... 42

6.- Manejo de periféricos. ..... 43
6.1.- Registros de control y de estado. ..... 43
6.2.- Controlador de dispositivo (device driver) ..... 44
6.3.- Las librerías de periféricos. ..... 44

# 1.- Introducción.

A continuación se van a revisar unas nociones básicas de lo que entendemos por sistema empotrado tal y como lo vamos a encontrar y desarrollar a lo largo del curso.

## *1.1.- ¿Que es un sistema empotrado?*

Un sistema empotrado es una combinación de hardware de computadora y software, y quizás componentes mecánicos adicionales diseñados para realizar una función específica. Un buen ejemplo es un horno microondas. En casi cada casa hay uno, y decenas de millones de ellos se usan cada día, pero pocos se dan cuenta que un procesador y software están relacionados con la preparación de su comida.

Este tipo de sistemas son muy diferentes de los ordenadores personales de que disponemos. Estos dispositivos también están compuestos de hardware de computadora, software y componentes mecánicos (unidades de disco, por ejemplo). Sin embargo, un ordenador personal no ha sido diseñado para realizar una función específica. En vez de esto, es capaz de llevar a cabo cosas muy diferentes. Hay personas que emplean el término "computadora de propósito general" para hacer esta distinción clara. Tal como se entrega, una computadora de propósito general está en blanco; el fabricante desconoce lo que el cliente va a hacer con ella. Un cliente puede usarlo como servidor de ficheros en una red de datos, otro lo puede usar exclusivamente para jugar a juegos de ordenador y otro puede emplearlo para escribir una novela.

Con frecuencia, un sistema empotrado es un componente dentro de otros sistemas más grandes. Por ejemplo, los coches modernos contienen varios sistemas empotrados. Uno de ellos controla los frenos anti-bloqueo, otro monitoriza y controla las emisiones del vehículo, y un tercero muestra información en el salpicadero. En algunos casos, estos sistemas empotrados están conectados por algún tipo de red de comunicación, pero esto ciertamente no es una necesidad.

Con el riesgo posible de confundir al alumno, es importante puntualizar que un ordenador de propósito general está a su vez compuesto de numerosos sistemas empotrados. Por ejemplo, mi ordenador consta de teclado, ratón, tarjeta de video, modem, disco duro, disquetera y tarjeta de sonido cada uno de los cuales es un sistema empotrado. Cada uno de estos dispositivos contiene un procesador y software y está diseñado para realizar una función específica. Por ejemplo, el modem está diseñado para enviar y recibir datos digitales sobre línea telefónica analógica. Todos los demás dispositivos pueden resumir su función en una solo línea de texto.

Si un sistema empotrado ha sido diseñado bien, la existencia del procesador y software puede pasar completamente inadvertida al usuario del dispositivo. Este es el caso de un horno microondas, un video, un reloj con alarma. En algunos casos, incluso sería posible construir un dispositivo equivalente que no contenga el procesador y el software. Esto se haría reemplazando la combinación con un circuito integrado a medida que realice las mismas funciones en hardware. Sin embargo, se pierde gran cantidad de flexibilidad cuando un diseño se "cablea" de esta forma. Es mucho más

sencillo y barato cambiar unas pocas líneas de software que rediseñar un trozo de hardware a medida.

## Historia y futuro

Dada la definición de sistema empotrado, el primer sistema de este tipo posiblemente no pudo aparecer antes de 1971. Este fue el año en que Intel introdujo el primer microprocesador del mundo. Este chip, el 4004, fue diseñado para usarse en una línea de calculadoras de oficina producidas por la compañía Japonesa Busicom. En 1969, Busicom pidió a Intel que diseñara un juego de circuitos integrados, uno para cada uno de sus nuevos modelos de calculadoras. El 4004 fue la respuesta de Intel. En lugar de diseñar hardware a medida para cada calculadora, Intel propuso un circuito de propósito general que pudiese ser usado en toda la línea de calculadoras. El procesador de propósito general fue diseñado para leer y ejecutar un juego de instrucciones almacenados en un chip de memoria externo. La idea de Intel fue que el software daría a cada calculadora un juego único de funciones.

El microprocesador tuvo un éxito tremendo, y su empleo se incrementó y se afianzó en la siguiente década. Las primeras aplicaciones empotradas incluyeron sondas espaciales no tripuladas, semáforos computerizados y sistemas de control de vuelo. En los años 80, los sistemas empotrados se introdujeron en la vida cotidiana. Muchos de los aparatos electrónicos de nuestras cocinas, salas de estar y lugares de trabajo son sistemas empotrados.

Parece inevitable que el número de sistemas empotrados continúe incrementándose.

## Sistemas en tiempo real

Una subclase de sistemas empotrados que merece la pena comentar en este momento. Como se definen comúnmente, un sistema en tiempo real es un sistema computador al que se le imponen restricciones de tiempo. En otras palabras, un sistema en tiempo real está especificado parcialmente en términos de su habilidad para realizar ciertos cálculos o decisiones de forma temporizada. Estos importantes cálculos se dice que tienen plazos para su terminación. Y, para todos los propósitos prácticos, un plazo no cumplido resulta tan malo como una respuesta incorrecta.

El problema de qué ocurre con un plazo no cumplido es crucial. Por ejemplo, si el sistema de tiempo real es parte de un sistema de control de un aeroplano, es posible que las vidas de los pasajeros corran peligro por un solo plazo no cumplido. Sin embargo, si en su lugar el sistema está involucrado en comunicaciones por satélite, el daño puede limitarse a un paquete de datos inservible. Cuanto más severas sean las consecuencias, más probable es que el plazo pueda considerarse “crítico” y, así, el sistema pueda considerarse crítico de tiempo real. Los sistemas de tiempo real contrapuestos a los críticos (hard) son los “blandos” (soft), que tienen plazos menos importantes.

Todo lo que se explica en este curso se aplica también a los diseños de sistemas en tiempo real. Sin embargo, los diseñadores de estos sistemas deben ser más diligentes en su trabajo. Deben garantizar que el funcionamiento va a ser correcto en software y hardware bajo todas las posibles circunstancias. Y, en el caso que vidas humanas

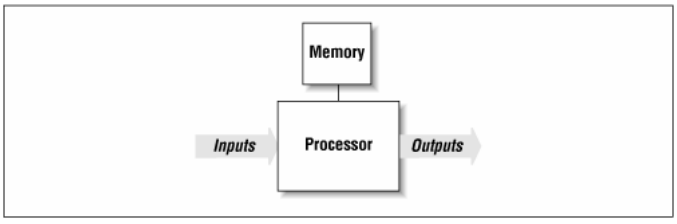
dependan del funcionamiento correcto del sistema, esta garantía debe certificarse mediante cálculos de ingeniería y documentos descriptivos.

## *1.2.- Variaciones del tema*

A diferencia del software diseñado para los ordenadores de propósito general, el software empotrado no puede normalmente ejecutarse en los sistemas empotrados sin modificar ligeramente. Esto es principalmente debido a la increíble variedad del hardware subyacente. El hardware de cada sistema empotrado está ajustado específicamente a la aplicación, para mantener el coste del sistema bajo. Como resultado, la circuitería innecesaria se elimina y los recursos hardware se comparten en la medida de lo posible. En esta sección aprenderemos cuales son las características hardware comunes a todos los sistemas empotrados y porque hay tanta variación.

Por definición todos los sistemas empotrados contienen un procesador y software, pero que otras características tienen en común. Ciertamente, para tener software, debe haber un lugar para almacenar el código ejecutable y el almacenamiento temporal para la manipulación de datos en tiempo de ejecución. Estos toman la forma de ROM y RAM, respectivamente, cualquier sistema empotrado tendrá algo de cada una. Si solo se necesita una pequeña cantidad de memoria, puede estar contenida en el mismo chip que el procesador. En otro caso, uno o ambos tipos de memoria residirán en chips de memoria externos.

Todos los sistemas empotrados también contienen algún tipo de entradas y salidas. Por ejemplo, en un horno microondas las entradas son los botones del panel frontal y las sondas de temperatura, y las salidas son el display y la radiación de microondas. Casi siempre las salidas del sistema empotrado son función de sus entradas y varios factores adicionales (tiempo transcurrido, temperatura actual, etc). Las entradas al sistema normalmente toman la forma de sensores y sondas, señales de comunicación, o mandos de control y botones. Las salidas son normalmente displays, señales de comunicación o cambios en el mundo físico.




Con la excepción de estas características comunes, el resto de hardware empotrado es único. Esta variación es el resultado de varios criterios de diseño en competencia. Por ejemplo, si el sistema debe tener un costo de producción menor de 10eur, otras cosas como la potencia de procesamiento y la fiabilidad del sistema deben sacrificarse para conseguir este objetivo.

Por supuesto, los costes de producción es una de las posibles restricciones bajo la que trabaja todo diseñador de sistemas empotrados. Otros requerimientos de diseños comunes incluyen los siguientes:

- **Potencia de procesamiento.**
  La cantidad de potencia de procesamiento necesaria para realizar el trabajo. Una forma común de comparar potencia de procesamiento es la tasa MIPS (millones de instrucciones por segundo). Si dos procesadores tienen tasas de 25MIPS y 40MIPS, el último será el más potente de los dos. Sin embargo, se deben considerar otras características importantes del procesador. Una de las cuales es el tamaño de los registros internos, que se encuentra en el rango 8 a 64 bits. Los ordenadores de propósito general emplean procesadores de 32 y 64bits exclusivamente, pero los sistemas empotrados todavía se construyen con procesadores más antiguos y baratos de 4, 8 y 16 bits.

- **Memoria.**
  La cantidad de memoria (ROM y RAM) necesaria para mantener el software ejecutable y los datos que se manipulan. Aquí el diseñador de hardware debe realizar una estimación correcta y preparase para incrementar o decrementar la cantidad real a medida que el software se va desarrollando. La cantidad de memoria necesaria puede afectar también la selección del procesador. En general, el ancho de registro de un procesador establece el limite superior de la cantidad de memoria a la que puede acceder (un registro de 8 bits solo puede seleccionar una de 256 posibles direcciones de memoria). Procesadores con ancho de palabra pequeño dispondrán de técnicas para aumentar el espacio direccionado.

- **Costes de desarrollo**
  El costo del hardware y del proceso de diseño de software. Este es fijo, solo se tiene en el proceso de diseño, luego para producciones de gran volumen no se suele tener en cuenta, o bien en el caso de pequeño numero de unidades producidas suele ser el coste a considerar.

- **Número de unidades**
  El compromiso entre costos de producción y costos de desarrollo se ve afectado principalmente por el número de unidades producidas y vendidas. Por ejemplo, normalmente resulta deseable desarrollar uno mismo los componentes hardware para un producto de bajo volumen de producción.

- **Tiempo de vida esperado**
  Cuanto tiempo de promedio el sistema va a continuar su función. Un mes, un año o una década. Esto afecta a muchas decisiones de diseño, desde la selección de componentes hardware a cuanto puede costar el sistema de desarrollar y producir.

- **Fiabilidad**
  ¿Cómo de fiable debe ser el producto final? Si se trata del juguete de un niño, no debe funcionar siempre correctamente, pero si forma parte de un coche, debe hacer lo que se supone en todo momento.

Además de estos requisitos generales, existen requisitos funcionales del sistema. Estos hacen que el sistema sea único.

La tabla siguiente ilustra el rango de posibles valores para cada uno de los requisitos previos al diseño. Estos son solo estimaciones y no deben tenerse en cuenta demasiado en serio. En algunos casos, dos o más criterios están relacionados. Por ejemplo, el incremento en la potencia de procesamiento lleva a un incremento en el coste de producción.

| Criterio | Bajo | Medio | Alto |
|---|---|---|---|
| Procesador | 4 u 8 bits | 16 bits | 32 o 64 bits |
| Memoria | < 16 KB | 64 KB a 1MB | > 1MB |
| Costo de desarrollo | < 100K eur | 100K a 1M eur | > 1M eur |
| Costo de producción | < 10 eur | 10 a 1K eur | > 1K eur |
| Núm de unidades | < 100 | 100 – 10.000 | > 10.000 |
| Tiempo de vida | Días, semanas o meses | Años | Décadas |
| Fiabilidad | Puede fallar | Debe ser fiable | Debe ser a prueba de fallos |

## *1.3.- C: el común denominador*

Una de las pocas constantes a través de todos estos sistemas es el empleo del lenguaje de programación C. Más que ninguno, C ha llegado a ser el lenguaje de los programadores de microcontroladores. Esto no ha sido siempre así, y no seguirá así siempre. Sin embargo, en este momento, C es la cosa más cercana a un estándar en el mundo de la programación de sistemas empotrados. Vamos a explicar porque C ha llegado a ser tan popular y porque lo hemos escogido para este curso.

Debido a que el desarrollo con éxito de software está tan relacionado con la selección del mejor lenguaje para un proyecto determinado, resulta sorprendente que un lenguaje se haya demostrado apropiado tanto para procesadores de 8 como de 64bits; en sistemas con bytes, kilobytes y megabytes de memoria, y para equipos de desarrollo que consistan de uno a una docena de personas. Precisamente en este rango de proyectos es donde C ha prosperado.

Por supuesto que C tiene ventajas. Es pequeño y simple de aprender, hay compiladores para casi cada procesador en uso hoy día, y hay una gran cantidad de experimentados programadores de C. Además, C tiene el beneficio de la independencia del procesador, que permite a los programadores concentrarse en los algoritmos y problemas a resolver, en lugar de los detalles de una arquitectura de procesador. Sin embargo, algunas de estas ventajas se aplican a otros lenguajes de alto nivel. Entonces porque C se emplea tanto frente a otros lenguajes?

Quizás la mayor ventaja de C es que es un lenguaje de alto-nivel orientado hacia el bajo nivel. Como veremos a lo largo del curso, C ofrece a los programadores de sistemas

empotrados un alto grado de control directo del hardware sin sacrificar los beneficios de los lenguajes de alto nivel. La naturaleza de bajo nivel de C fue una clara intención de sus creadores.

Pocos lenguajes de alto nivel populares pueden competir con C en la producción de código compacto y eficiente para casi todos los procesadores. Y de estos, solo C permite a los programadores interactuar con el hardware subyacente tan fácilmente.

## *1.4.- Unas palabras acerca el Hardware*

Por definición todos los libros de programación sobre el tema que nos ocupa deben incluir ejemplos. Normalmente, estos ejemplos están seleccionados de forma que puedan experimentarse fácilmente por los lectores interesados. Esto significa que los lectores deben tener acceso a las mismas herramientas de software y plataformas de hardware empleadas por el autor. Desafortunadamente, en el caso de la programación de sistemas empotrados esta afirmación no es realista. Simplemente no tiene sentido ejecutar ninguno de los programas de ejemplo en las plataformas disponibles a la mayoría de los lectores, PCs, Macs y estaciones de trabajo UNIX.

Incluso seleccionar una plataforma de microcontrolador estándar es difícil. Como ya hemos comentado, no hay ninguna cosa como sistema empotrado “típico”. Una vez seleccionado el hardware, la mayoría de los lectores no tienen acceso a ello. Pero a pesar de este problema bastante importante, creemos que es importante seleccionar una plataforma hardware para emplearla en los ejemplos. Para hacer tal cosa, espero hacer los ejemplos consistentes y, así, los comentarios al completo más claros.

Para ilustrar la mayoría de los tópicos posibles con un simple trozo de hardware, he creído conveniente seleccionar una plataforma de tamaño pequeño, para que sea fácil y barata de montar para el alumno. Como este hardware no se ha encontrado en el comercio con las características adecuadas, se ha decidido diseñar un pequeño entrenador basado en un microcontrolador de microchip, el PIC. Este hardware consiste en un microntrolador de 8 bits, con 364 bytes de RAM y 8K palabras de ROM, junto con una serie de periféricos que permiten desarrollar el temario del curso. La placa se ha diseñado y se le ha dado el nombre II2, relacionado con la asignatura. En la página web de la asignatura (http://www.uhu.es/manuel_sanchez) se puede descargar el sencillo esquema, la PCB, la lista de componentes y la disposición de éstos. Se puede obtener más información en esta misma página.

Si dispones del hardware de referencia, serás capaz de trabajar con los ejemplos del libro exactamente como están presentados. En otro caso, necesitaras portar el código ejemplo a otra plataforma empotrada a la que tengamos acceso. Hacia este fin, cada esfuerzo se ha hecho para hacer los programas ejemplo tan portables como sea posible. Sin embargo, el lector debe tener en cuenta que el hardware de cada sistema empotrado es diferente y que algunos de los ejemplos podrán no tener sentido con ese hardware. Por ejemplo, no tendrá sentido portar el driver de memoria Flash presentado en el capitulo correspondiente a una placa que no tenga dispositivos de memoria Flash.

# 2.- Primer programa en C.

En este capitulo nos sumergiremos en la programación de sistemas empotrados a través de un ejemplo. El programa que vamos a probar tiene un espíritu similar al ejemplo "Hola, Mundo" que se encuentra al comienzo de otros libros de programación en C. A medida que comentemos el código, encontraremos justificación para la selección del programa en particular y apuntaremos las partes de él que dependen del hardware final. Este capitulo solo contiene el código fuente para el primer programa. Discutiremos como crear el ejecutable y ejecutarlo en los dos capítulos siguientes.

## *2.1.- Hola Mundo!*

Un ejemplo tan usado como este podría parecer un poco aburrido. Pero ayuda a los lectores a evaluar la dificultad o facilidad con la que se pueden escribir simples programas en el entorno de programación. En este sentido "Hola, mundo" cumple su función como una medida útil de los lenguajes de programación y plataformas. Desafortunadamente, mediante esta medida, los sistemas empotrados son las plataformas más complicadas para que trabajen con ellas los programadores. En algunos sistemas empotrados, podría ser imposible implementar el programa "Hola, mundo". Y en esos sistemas capaces de soportarlo, la impresión de cadenas de texto resulta normalmente un punto final en lugar de un comienzo.

Lógicamente, la suposición de que existe algún tipo de dispositivo de salida en el que se pueden imprimir cadenas de texto resulta crucial para que el ejemplo funcione. Una ventana de texto en el monitor del usuario suele servir para esto. Pero la mayoría de sistemas empotrados carecen de monitor o dispositivo de salida similar. Y los que tienen uno necesitan una pieza de software especial, denominada driver de pantalla, bastante complicada para comenzar a explicarla. Por tanto, resulta mejor escribir un programa pequeño y portable para esta primera vez.

Los programas empotrados deben ser auto-explicativos. Debemos comenzar nuestro nuevo proyecto con la suposición de que los demás no conocen nada acerca del funcionamiento de nuestro sistema, solo la sintaxis del lenguaje. Incluso podrían no tener acceso a las rutinas básica de librerías. Estas son las funciones auxiliares como "printf" y "scanf" que la mayoría de los programadores dan por supuesto que se pueden emplear en C. De hecho, las rutinas de librería son tan estándar como parte del lenguaje y la sintaxis básica. Sin embargo, esta parte del estándar resulta más complicada de transportar a todas las posibles plataformas hardware de diferentes fabricantes y normalmente se ignoran por los fabricantes de software de desarrollo de sistemas empotrados.

No encontraremos un programa completo en este capitulo. En su lugar supondremos que solo disponemos de la sintaxis básica de C para nuestro primer ejemplo. A medida que progresemos, añadiremos gradualmente sintaxis de C, rutinas de librería estándar, y el equivalente a un dispositivo de salida de caracteres a nuestro repertorio. Luego, en el capitulo correspondiente, finalmente implementaremos un programa "Hola, Mundo". En ese momento estaremos dispuestos a experimentar para conseguir ser un experto en el campo de la programación de sistemas microcontroladores.

## *2.2.- .El LED parpadeante*

Cada sistema empotrado que encontraremos a lo largo de la carrera profesional tendrá al menos un LED que puede controlarse por software. Por lo que el sustituto para el programa “Hola, mundo” es uno que hace parpadear un led a una velocidad de 1 Hz (un ciclo completo por segundo). Normalmente, el código necesario para encender y apagar un LED está limitado a unas pocas líneas de C o ensamblador, por lo que hay poco lugar para que sucedan errores de programación. Y debido a que casi todos los sistemas empotrados tienen un LED, los conceptos comentados son bastante portables.

La estructura del programa de LED parpadeante es la siguiente. Esta parte del programa depende del hardware. Sin embargo se emplean las funciones dependientes del hardware *toggleLed* y *delay* para cambiar el estado del LED y manejar la temporización respectivamente.

El registro de funcion especial ADCON1 controla la asignación de las patillas del puerto que se va a usar (PORTA) para que se conecten de forma digital y se puedan usar como E/S.

El PIC dispone de un registro para cada puerto denominado TRISX, donde X es el nombre del puerto (A,B,C) y que controla cada bit asignado a una patilla, de forma que si este bit vale cero se comporta la patilla como salida y si vale uno se comporta como entrada.

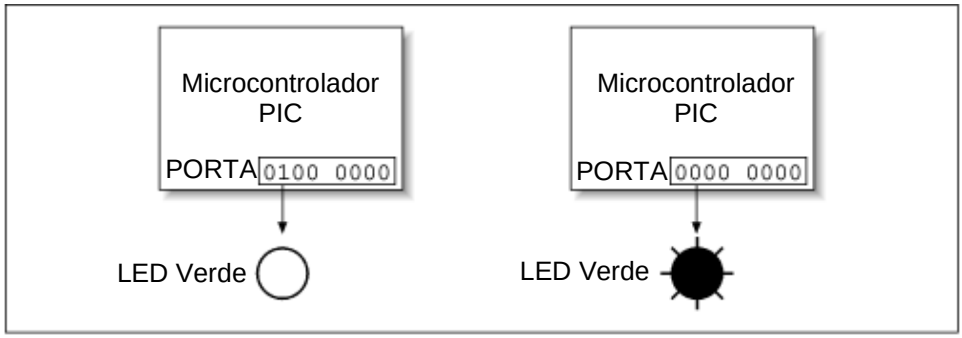
```
#include <pic.h>
#define LED_GREEN 0x40  /* Led verde se controla mediante el bit 6 */

/************************************************************************
 *
 * Funcion:    main()
 *
 * Descripcion: Parpadea LED verde una vez por segundo.
 *
 * Notas:       El bucle no depende del hardware.  Sin embargo,
 *              depende de dos funciones dependientes del hardware.
 *
 * Devuelve:    Contiene un bucle infinito.
 *
 ************************************************************************/
void main(void)
{
    ADCON1=0x07;              // Inicializa los puertos digitales
    TRISA=0x13;               // Inicializa el Puerto como E/S
    while (1)
    {
        toggleLed(LED_GREEN);      /* Cambia el estado del LED.  */
        delay(500);                /* Pausa 500ms.     */
    }

}   /* main() */
```

### 2.2.1.- toggleLed.

En el caso de un sistema basado en el PIC y en la placa en concreto que disponemos en el laboratorio esta dispone de un LED verde. El estado del LED se controla mediante un bit en un registro denominado Puerto A de E/S (PORTA). Este registro está colocado en el mismo chip que la CPU. Cada uno de los ocho bits del puerto está asociado con el voltaje de una de las patillas de E/S. Por ejemplo, el bit 6 controla el voltaje que va al LED verde:

```
#define LED_GREEN 0x40  /* Led verde se controla mediante el bit 6 */
```

Modificando este bit es posible cambiar el voltaje de la patilla externa, y de esta forma, el estado del LED verde. Cuando el bit 6 de PORTA es 1 el LED está apagado, cuando es 0 el LED está encendido.




El registro PORTA está colocado en una región especial de memoria denominada registros de función especial. El acceso a esta porción del espacio se realiza con las instrucciones de ensamblador del PIC SETB, similares a las MOV del 8051 empleando direccionamiento directo.

Para implementar la rutina toggleLed en la placa específica II2 no emplea ninguna rutina de librería como se muestra abajo. El algoritmo real lee el contenido directamente del registro PORTA, cambia el bit que controla el LED de interés, y escribe el nuevo valor de nuevo en el registro.

```
#include <pic.h>    /* Fichero de cabecera para incluir la */
                       /* definición del puerto A */

/*********************************************************************
 *
 * Funcion:    toggleLed()
 *
 * Descripcion: Cambia el estado de uno o varios LEDs.
 *
 * Notas:       Esta función es específica a la placa II2.
 *
 * Devuelve:     Nada.
 *
 ********************************************************************/
```

```
void
toggleLed(unsigned char ledMask)
{
        unsigned char bit_a_cambiar;

        /* miro si estaba a cero o a uno */
        bit_a_cambiar=(PORTA&ledMask);

        if (bit_a_cambiar)
                PORTA=PORTA&(~ledMask);
                /* si es uno pongo a cero */
        else
                PORTA=PORTA|ledMask;
                /* si es cero pongo a uno */
}   /* toggleLed() */
```

### 2.2.2.- delay

También necesitamos implementar un retardo de medio segundo (500ms) entre cambios del LED. Esto se hace mediante una espera activa dentro de la rutina delay más abajo. Esta rutina acepta la longitud del retraso pedido, en milisegundos, como único parámetro. Luego multiplica este número por la constante CYCLOS_POR_MS para obtener el número total de iteraciones del bucle while para generar el retraso pedido.

```
/*********************************************************************
 *
 * Funcion:    delay()
 *
 * Descripcion: Espera activa durante los ms pedidos.
 *
 * Notas:       El numero de ciclos por milisegundos de decremento
 *              y test fue determinado mediante prueba y error.  Este
 *              valor depende del tipo de procesador y velocidad
 *
 * Devuelve:     Nada.
 *
 ********************************************************************/
void
delay(unsigned int nMilliseconds)
{
    #define CICLOS_POR_MS 260 /* Ciclos de decremento y test. */

    unsigned long nCycles = nMilliseconds * CICLOS_POR_MS;

    while (nCycles--);

}   /* delay() */
```

Esta constante específica CICLOS_POR_MS representa el número de ciclos de decremento y test que el procesador puede realizar en un milisegundo. Para determinar este número empleamos prueba y error. Hacemos un cálculo aproximado (pienso que estará alrededor de 50), luego hacemos el resto del programa, lo compilamos y los probamos. El LED parpadeará a una velocidad más grande de 1Hz. Usamos nuestro reloj de pulsera para hacer una serie de pequeños cambios a la constante

CICLOS_POR_MS hasta que la velocidad de parpadeo esté tan cerca de 1 Hz como podamos.

Esto es todo. Todo lo necesario para el programa de parpadeo. Las tres funciones main, toggleLed y delay hacen toda la tarea. Si queremos portar este programa a otro sistema empotrado, debemos leer la documentación que viene con el hardware y volver a reescribir toggleLed como sea necesario y cambiar el valor de CICLOS_POR_MS. Por supuesto, todavía necesitamos comentar como compilar y ejecutar este programa. Veremos esto en los siguientes capítulos. Pero primero, vamos a hablar un poco de bucles infinitos y su papel en los sistemas empotrados.

## *2.3.- El papel del bucle infinito*

Una de las diferencias fundamentales entre los programas desarrollados para sistemas empotrados y los escritos para otras plataformas de ordenador es que los programas empotrados casi siempre acaban con un bucle infinito. Normalmente, este bucle rodea una parte grande de la funcionalidad del programa como en el programa anterior del LED. El bucle infinito es necesario porque el trabajo del software empotrado nunca se termina. Va a estar funcionando hasta el fin del mundo o que la placa se resetee, lo que ocurra primero.

Además, la mayoría de sistemas empotrados solo tienen un trozo de código ejecutándose. Y aunque el hardware es importante, no funcionarían en absoluto sin el software empotrado. Si el software deja de funcionar, el hardware no sirve de nada. Por eso, las partes funcionales de un programa empotrado casi siempre se rodean de un bucle infinito que asegure que se ejecutan siempre.

Este comportamiento es tan común que no merece la pena mencionarlo. Aunque algunos de vosotros podríais ver encenderse el LED una sola vez porque hayáis olvidado incluir en el bucle infinito las llamadas a toggleLed y delay.

## *2.4.- Documentación en el desarrollo de software.*

La documentación en el desarrollo de software es muy importante, pero resulta más importante en los sistemas empotrados, pues al carecer de dispositivos estándar resulta necesario entender la interfaz única de nuestro sistema a través de una documentación adecuada.

Un documento que frecuentemente se descuida es la *interfaz hardware-software*. Esta consiste simplemente en una tabla donde aparece descrita la dirección de memoria de cada dispositivo periférico con el significado de cada bit del puerto de E/S definido por esa dirección de memoria.

Cuando pasamos la documentación a otro ingeniero o la repasamos pasados unos meses nos volvemos locos buscando donde están, como se emplean y que significado tienen los puertos de E/S que emplea nuestro sistema para comunicarse con el mundo exterior. Esta información puede y debería residir en la cabecera (al comienzo, antes de la función main) de nuestro programa.

Para la representación de la funcionalidad del software generalmente se recurre a los *diagramas de Bloques*, también denominados *organigramas*.

Los diagramas de flujo no son dibujos detallados del programa, si no en su lugar una visión rápida del método seguido en su solución. Como cualquier código de un programa que no puede ser más largo que una página, un diagrama debe estar también contenido en una sola página. El principal requisito de un diagrama de flujo es que sea fácilmente entendible por cualquier persona. Cuando se hagan tan complejos que no se puedan colocar en una página es el momento de simplificarlos (y el programa que representa el diagrama) haciendo subrutinas.

Habrá que expandir los detalles de estas subrutinas en diagramas separados en otras páginas. De esta forma, el diagrama de flujo principal siempre dará una visión general de las piezas del programa. Si necesitamos más detalle sobre el método de solución, nos dirigiremos a la subrutina apropiada.

Todos los diagramas deberían usar nombres funcionales y generalmente no referirse a nombres de variables específicas.

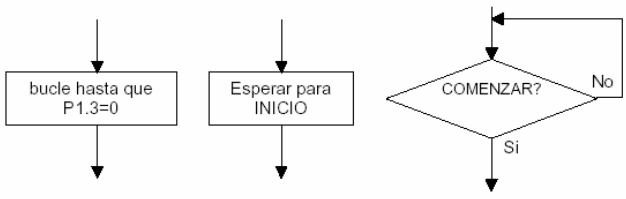
En los siguientes apartados se observan los bloques más empleados y su relación con el código fuente C.

# Saltos

Los microcontroladores demuestran su utilidad cuando toman la información del exterior y la emplean para tomar decisiones sobre la tarea que tengan que realizar:

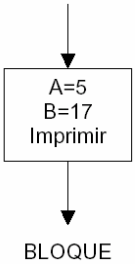
- Dependiendo de la condición de un pulsador, activan una válvula de control o no.
- Si esta operación se ha realizado 22 veces, avanza y realiza la siguiente operación.
- Se mantiene comprobando la señal que informa que el chip de síntesis vocal puede tomar el código de la siguiente palabra.

Todos estos ejemplos son ***decisiones*** que realiza un microcontrolador de manera rutinaria. Basado en el test de decisión, el flujo del programa realizará un ***bucle*** (vuelta hacia atrás) o ***salto*** (avanzar en una de las posibles direcciones).



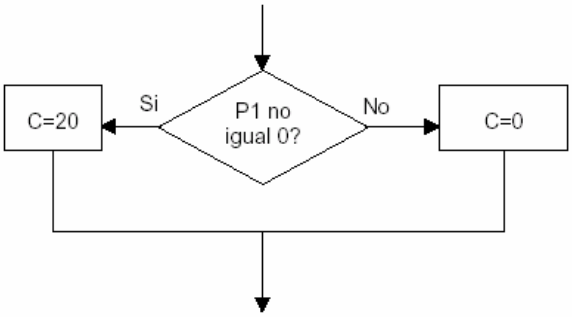

C es un lenguaje estructurado, hay reglas rígidas que prohíben romper el flujo del programa de forma arbitraria. Un lenguaje estructurado nunca permite saltar dentro de otra función sin guardar y restaurar la pila y cualquier otro registro pertinente. Empleando programación estructurada, aparte del caso especial de las interrupciones, no podemos corromper la pila con ningún conjunto de instrucciones aceptables.

El elemento básico de un lenguaje estructurado es el bloque. Es un trozo de programa donde el flujo entra solo por un lugar y lo abandona solo por otro lugar. No se puede entrar en la mitad del bloque o ejecutarlo parcialmente. El ejemplo siguiente muestra una estructura de bloque simple en C. Los tres programas ejemplo son variaciones del mismo bloque.




```
{ /* estructura básica de bloque */
a=5;
b=17;
print;
}
{ /* estilos alternativos de bloque */
A=5; B=17; print();
}
{A=5; B=17; print();}
```

Las diferencias son relativas al hecho que C no tiene en cuenta los saltos de línea. Las sentencias individuales de C no tienen que estar en líneas separadas. Podemos hacer de esta forma los listados del programa más cortos o más largos. Esto puede servir para agrupar varias expresiones cortas cuando tengan relación entre ellas.

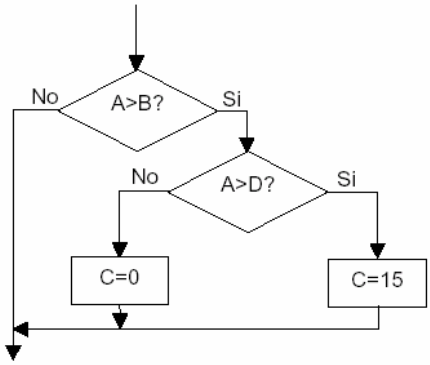



```
/* decisión if/else */
if (P1!=0) c=20;
else c=0;
```

# Bifurcación

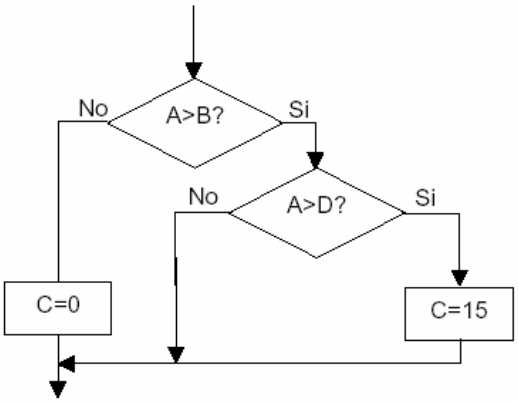
Una decisión básica o bifurcación en el flujo del programa lo representa el bloque ***if/else***. El else es opcional, y las expresiones pueden ser bloques ellos mismos. En C, la estructura if/else es sencilla.

C permite realizar asignaciones dentro de la expresión de test, esto se denomina *asignación implícita*. Aquí es donde ++i opuesto a i++ puede resultar significativo, puesto que esto determina si la variable se incrementa antes o después del test.




Versiones 1 y 2

Los bloques else pueden anidarse, como se muestra a continuación. El anidamiento de los if enlaza el else con el if más reciente a no ser que la estructura de bloques lo defina de otra forma. El primer y segundo trozos de código en el siguiente ejemplo son idénticos, aunque la indentación sugiera que el segundo trozo de código ponga C=0 si A no es mayor que B. Si esto es lo que queremos, entonces, en el tercer trozo de código colocamos el bloque necesario {} para que el else se aplique al primer if.




Versión 3

```
/* bloques if/else anidados */
/* versiones 1, 2 y 3 */
if (a>b) {
if (a>d) c=15;
else c=0;
}
/* ver2 */
if (a>b)
if (a>d) c=15;
else c=0;
/* ver3 */
if (a>b) {
if (a>d) c=15;
}
else c=0;
```

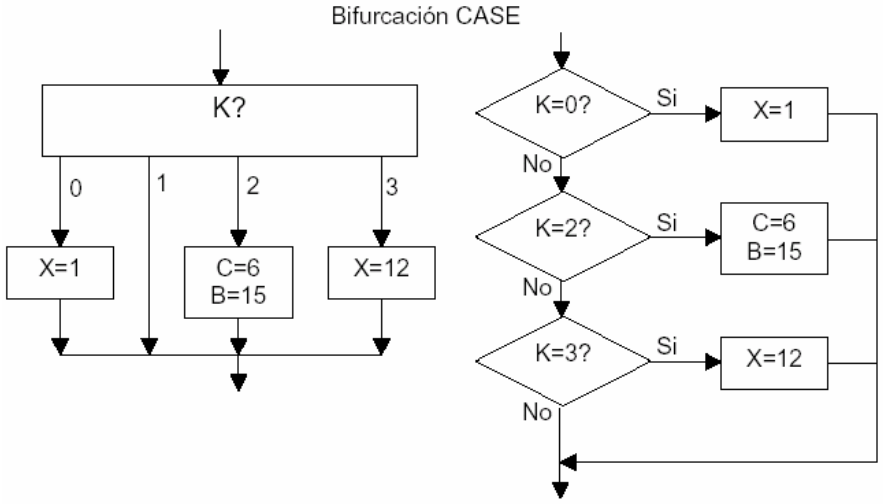
## Operador Condicional

Una expresión única en C es el operador condicional. Es una abreviatura para una decisión if/else donde las dos opciones asignan simplemente un valor diferente a una variable. Es un test donde la condición cierta asigna el primer valor y la condición falsa asigna el segundo valor.

```
/* operador condicional */
c = (a>d) ? 15 : 0;
```

## Switch

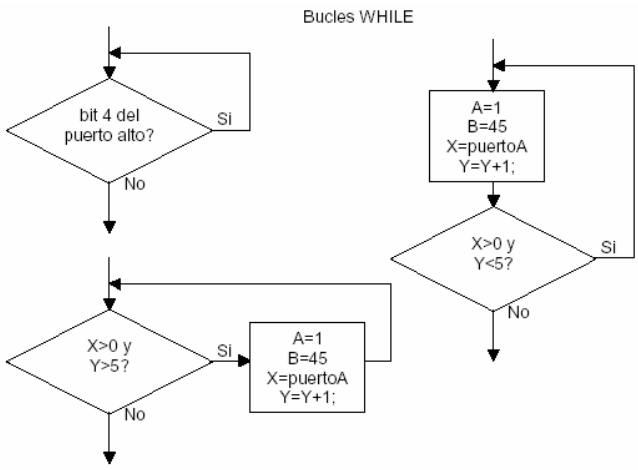
Además de las simples construcciones de bifurcación en C tenemos la construcción switch. Permite cambiar el flujo del programa en muchas formas basándose en el valor de una variable o expresión. Una cadena de construcciones if/else pueden realizar la misma función, pero un switch puede hacer un salto multivía más entendible. La forma más simple es cuando la bifurcación depende de un entero, pero puede depender también del valor de una expresión. No tenemos que listar todos los casos posibles, tenemos un caso por defecto, o cualquier caso no existente caerá en este grupo.




```
/* bifurcación case básica */
switch (k) {
case 0:
x=1;
break;
case 2:
c=6;
b=15;
break;
case 3: x=12;
break;
default: break;
}
```

## Bucle While

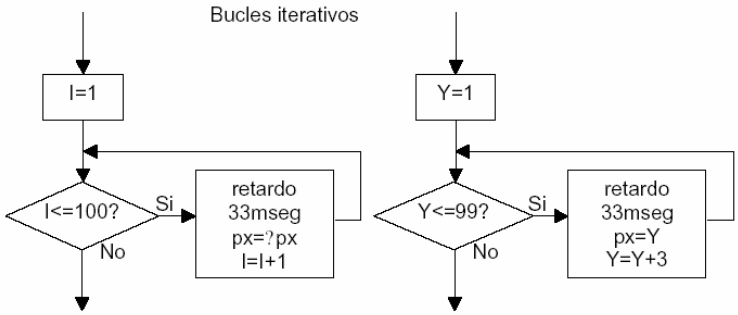
Una construcción de bucle es el bloque ***while***. El flujo del programa continúa con el bucle hasta que el test deje de ser cierto. Una forma emplea el test primero, entrando en el bloque solo si pasa el test. Si no, el flujo se salta el bloque y continúa con la primera sentencia tras el bloque. Una segunda forma, el bucle do...while(), realiza el test al final del bloque para decidir si volver atrás y hacerlo de nuevo o continuar. De esta forma, el bloque siempre se ejecuta al menos una vez.




```
/* bucle while simple (vacio) */
while ((P1 & 0x10)==0);
/* bucle while normal */
while (x>0 && y++==5) {
a=1;
b=45;
x=P1;
}

/* bucle do while */
do {
a=1;
b=45;
x=P1;
}
while (x>0 && y++==5);
```

# Bucle Iterativo

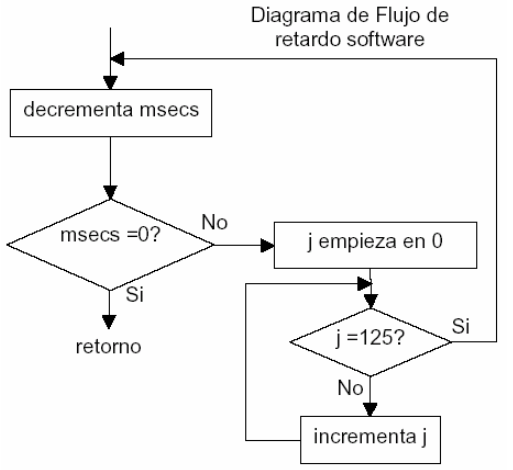
Una segunda estructura muy usada en C es el bucle ***iterativo***. Emplea constantes, variables e incluso expresiones complejas para controlar el número de veces que se ejecuta el bucle. La instrucción tiene tres partes. La primera es la ***expresión inicial***. Esto normalmente asigna un valor numérico, pero puede ejecutar cualquier expresión cuando entramos por primera vez en el bucle. Luego tenemos la ***prueba para finalizar*** el bucle. Esto normalmente verifica que se cumple una determinada condición de un índice, pero podemos tener cualquier sentencia condicional cuyo fallo terminará el bucle. Finalmente, tenemos el ***incremento de índice***. Normalmente es positivo. Cada vez que se ejecuta el bucle incrementamos (o decrementamos) la variable índice. Esta parte puede tener cualquier operación o expresión que se realiza tras fallar la condición de salida antes de volver a ejecutar el bloque. Podemos hacer cosas casi incomprensibles con esta construcción.




```
/* estructuras de bucles iterativos */
for (i=1;i<=100;i++) {
retardo(33);
px=?px;
}
/* **** */
for (y=0;y<=99;y=y+3) {
retardo(33);
px=y;
}
/* **** */
for (da=inicio;estado==ocupado;leds=?leds) {
retardo(33);
}
```

# Ejemplo: retardo de tiempo.

Es común producir un retardo de tiempo mediante bucles anidados de forma que la ejecución de instrucciones consuma una cantidad de tiempo conocida. El siguiente ejemplo muestra estos retardos de tiempo. El tiempo en el bucle depende del número de ciclos de reloj y la frecuencia de reloj. La función de retardo aquí consume alrededor de un mseg para cada unidad que se le pasa. Pasando un valor de 50 tenemos un retardo de aproximadamente 50 x 100 = 5000 µseg = 5 mseg.




```
/* retardo mediante bucle software */
void msec(unsigned int x) {
unsigned char j;
while (x-- > 0) {
for (j=0;j<125;j++){;}
}
}
```

Si el diagrama es muy grande se realiza en varias hojas y se utilizan conectores numerados para enlazar las diferentes partes del diagrama. Sin embargo se aconseja realizar un diagrama pequeño o mediano que no ocupe más de un lado de un folio para que se pueda seguir con facilidad. El resto de funcionalidad del programa se debe colocar en las subrutinas cuyo diagrama de bloques se realiza en otras páginas de la documentación del sistema.

Estos diagramas también se emplean en otros campos de la ingeniería para representar procesos industriales o algoritmos. Un algoritmo se emplea en programación de ordenadores para describir las fases que se deben realizar para alcanzar un objetivo y en la documentación deben recogerse los algoritmos empleados en el sistema.

Un código fuente bien redactado debe ser autoexplicativo. No sólo deben tener comentarios que aporten información, si no que también el estilo debe ser correcto. Esto quiere decir que los nombres de las variables, de las constantes y de las rutinas deben en la medida de lo posible tener sentido y asociarse con su función en el sistema. Las constantes se suelen colocar en MAYUSCULAS, las variables en minúscula y las rutinas capitalizadas, es decir con la primera letra significativa en mayúscula, por ejemplo: LeerPuerto.

Aunque se pueden realizar diagramas muy complejos, hay que tener cuidado pues existen algunas limitaciones a la hora de implementar un diagrama complejo en código fuente C. Puede darse el caso de que para la implementación de algún diagrama con vueltas atrás in/condicionales sea necesario repetir partes del código C, pues recordemos que no se pueden emplear los saltos indiscriminados a partes del programa en los lenguajes estructurados como C. Por esto, se aconseja que las vueltas atrás que se coloquen en el diagrama hagan referencia a una estructura válida de C como while, do..while o for.

## *2.5.- Anatomía de un programa.*

Se pueden realizar programas básicos con estructuras muy sencillas. Vamos a ver dos ejemplos, un reloj digital realizado mediante dos estructuras de programación muy empleadas: atención a sensores e interrupciones.

### 2.5.1.- Atención a sensores.

En un programa empotrado que debe realizar ciertas tareas y atender al usuario o a sensores en tiempo real, se emplea una estructura basada en un bucle infinito de atención a sensores para simular una atención a estos sensores en tiempo real.

El bucle principal va testeando los sensores y si requieren atención se dispara la rutina de atención que corresponda. Para conseguir una atención a cada sensor lo más rápida posible las rutinas de atención se deben codificar de forma que sean lo más cortas y rápidas posibles y de esa forma que el programa vuelva al bucle principal donde pueda atender otro sensor.

En la figura se observa el diagrama de bloques básico de este sistema.

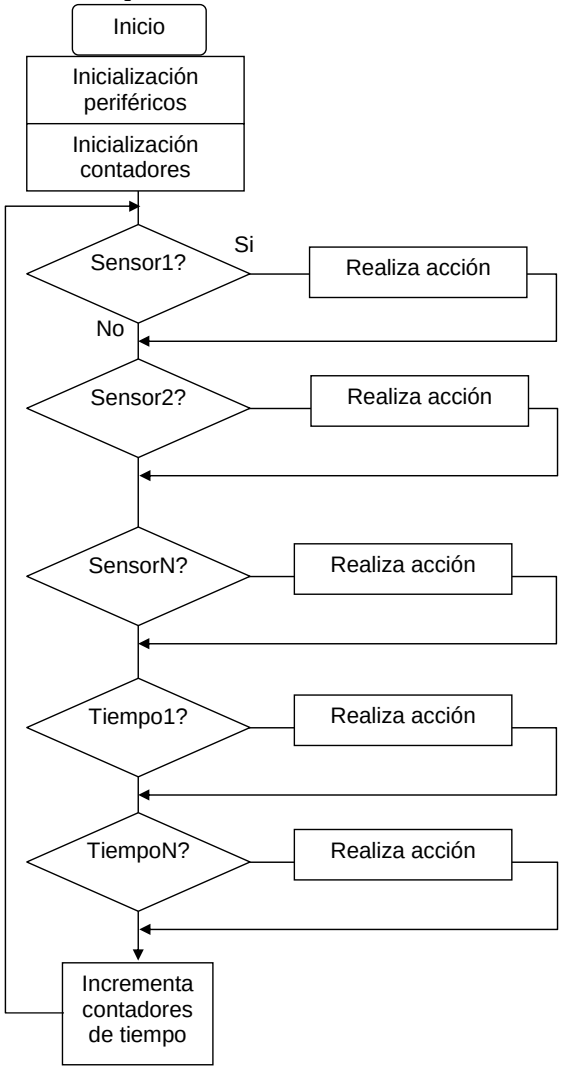



Diagrama de Flujo de programa tipo

El mayor problema con este método es que necesitemos implementar periodos de espera grandes para realizar alguna tarea en los actuadores. En ese caso si transcurridos 5 segundos tras detectar un cambio en un sensor debemos activar una salida, la implementación de esta espera empleando la rutina delay anterior provocaría que dejáramos de atender durante este tiempo el resto de sensores, lo que conocemos como *espera activa*. El sistema dejaría de funcionar correctamente durante estos cinco segundos.

Para evitar este problema, se define unos contadores de tiempo que se va a implementar en software mediante el empleo de variables auxiliares y de un retardo de tiempo corto en el bucle principal del sistema que va contando el tiempo pasado sin dejar de atender todos los sensores. Se debe añadir un testeo para la condición que el tiempo haya expirado como si se tratase de un sensor más. Además debe existir un mecanismo para activar y desactivar el temporizador software.

Como ejemplo tenemos el reloj digital implementado de esta forma. En el bucle principal se van probando todas las posibilidades, mientras que se espera un segundo para marcar el tiempo del contador.

En el peor de los casos se podría demorar la atención tanto como un segundo, lo que para algunas aplicaciones de tiempo real resulta inaceptable. Para estos casos se emplea el mecanismo de interrupciones o se emplea hardware especialmente diseñado. La rutina muestra_display, que no está implementada, mostrará en el display LCD el tiempo transcurrido en horas, minutos y segundos.

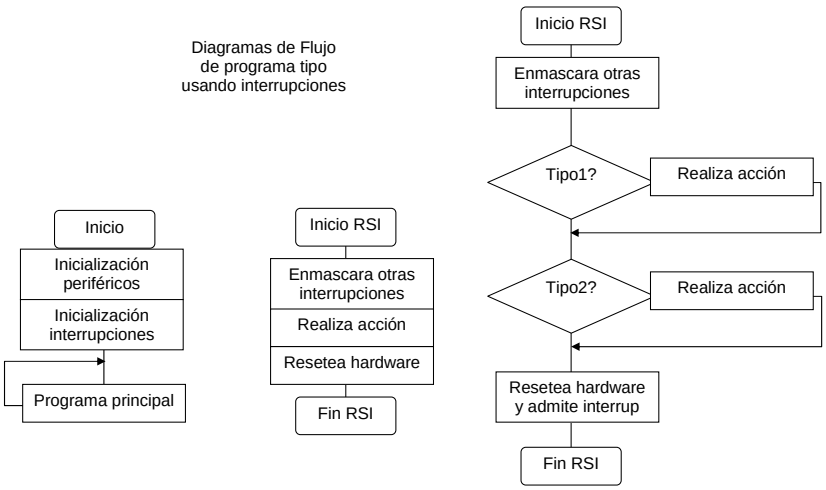
```
/* Reloj digital empleando temporización por bucle */

#include <16f876.h>

/* Función que introduce un retardo de un segundo aproximadamente */
void retardo(void) {
unsigned char i,j,k;

      for (i=0;i<100;i++)
            for (j=0;j<200;j++)
                  for (k=0;k<200;k++);
}


void main(void) {
unsigned char segundos,minutos,horas;

      /* Inicializa reloj */
      segundos=minutos=horas=0;

      while (1) {
            segundos++;
            if (segundos >= 60) { segundos=0;minutos++; }
            if (minutos >= 60) { minutos=0;horas++; }
            if (horas >= 24) horas=0;
            retardo();
            muestra_display(horas,minutos,segundos);
      }
}
```

## 2.5.2.- Interrupciones.

Para la atención a los sensores en tiempo real el mecanismo de las interrupciones resulta más útil, puesto que el retraso en atenderlos es del orden de microsegundos. Sin embargo, no podremos atender normalmente a todos los sensores por limitaciones en el número de interrupciones que se pueden manejar por el microcontrolador.

Se debe contar con una rutina de inicialización del sistema hardware de interrupciones y de una *rutina de servicio a la interrupción* (RSI) que será llamada cada vez que se produzca un cambio en el nivel lógico de la patilla de interrupción del sistema o bien se produzca una interrupción interna del microcontrolador debida a alguna situación interna (nuevo carácter recibido en la interfaz serie, contador expirado), dependiendo del hardware.

Para trabajar con interrupciones se recurre al sistema de diagrama de bloques de la figura:




El programa principal solo inicializa el sistema de interrupciones y se dedica a realizar otras tareas que deba en su bucle principal, tal como se comentó en el capitulo anterior.

La rutina de atención a la interrupción es la encargada de realizar la acción necesaria. Esta rutina debe ser corta, porque durante su ejecución la ejecución del programa principal se detiene para pasar a ejecutar la rutina de atención. No pueden realizarse llamadas al programa principal ni a rutinas que no sean *reentrantes* (las rutinas que tienen estado interno y puede ser que no funcionen al llamarse al mismo tiempo desde la RSI y del programa principal).

No deben emplearse variables que emplee el programa principal para realizar su tarea o variables que empleen las rutinas de librería para guardar su estado puesto que podrían hacer que el programa principal fallara o estas rutinas fallaran al ejecutarse simultáneamente trozos de código del programa principal y de la RSI. Sin embargo, hay variables especiales para comunicación entre la RSI y el programa principal que se emplean y resulta correcto que se compartan entre ambas. Cuando la acción que se debe realizar en atención a una interrupción no es corta (opuesto a cambiar un bit de un puerto de salida) o necesita rutinas especiales con estado que su implementación en la RSI podría comprometer el funcionamiento del programa lo que se hace es emplear *banderas*.

Las banderas no son más que indicadores implementados mediante variables compartidas entre la RSI y el programa principal que tienen como misión notificar al bucle del programa principal que ha tenido lugar una interrupción y es la manera correcta de hacerlo en lugar de utilizar mecanismos dependientes del hardware como la lectura de registros especiales por parte del programa principal. En algunos casos, como sucede con la arquitectura del PIC, se tiene solo una interrupción para atender a todos los periféricos y es necesario realizar un testeo del tipo de periférico que ha solicitado la interrupción dentro de la propia RSI para realizar el tratamiento apropiado.

El programa anterior realizado mediante interrupciones en el caso del 8051 muestra como el timer interno del 8051 dispara la interrupción cada 500ms y es entonces cuando se modifica el estado de la variable contador compartida con el programa principal. El programa principal tan solo se ocupa de mostrar por pantalla el valor actual del reloj.

```
#include <reg51.h>
#include <stdio.h>
#define TIEMPO 1000

#define bajo(x)  ((unsigned char)((x)&0x00FF))
#define alto(x) ((unsigned char)((x)>>8))
unsigned char segundos,minutos,horas;
unsigned int msec;

/* Función que introduce un retardo de un segundo aproximadamente */
void timer0() interrupt 1 {

      msec++;
      if (msec>=1000) {msec=0;segundos++;}
      if (segundos >= 60) { segundos=0;minutos++; }
      if (minutos >= 60) { minutos=0;horas++; }
      if (horas >= 24) horas=0;
      /* Reprogramar el timer */
      TH0 = alto(65535-TIEMPO);
      TL0 = bajo(65535-TIEMPO);
}

void main(void) {
      /* Inicializa puerto serie */
      SCON = 0x52; /* Configura comunicación serie modo 1 */
      TCON = 0x40;
      TMOD = 0x20; /* Timer 1 en Modo 2, GATE=0 y C/T=0 */
      TH1 = 0x0FD; /* Valor para 9600 baudios */

      /* Inicializa reloj */
      msec=segundos=minutos=horas=0;

      /* Inicializa timer 0 */
      TMOD |= 0x02;
      TH0 = alto(65535-TIEMPO);
      TL0 = bajo(65535-TIEMPO);
      TR0 = 1;  /* Pone en march timer 0 */
      ET0 = 1; /* Habilita int timer 0 */
      EA = 1; /* Habilitacion general de int */
      while (1) {
            printf("%02u:%02u:%02u\r",(unsigned     int)horas,(unsigned
            int)minutos,(unsigned int)segundos);
      }
}
```

# 3.- El proceso de Compilar, Linkar y Cargar.

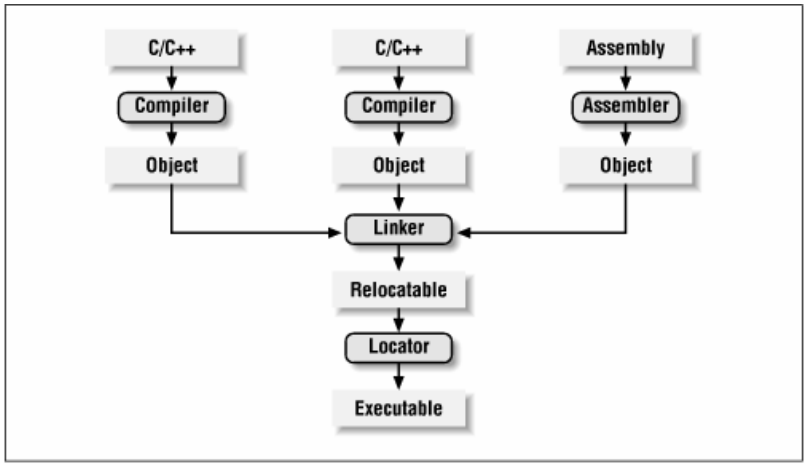
En este capitulo veremos los pasos en nuestro software para la ejecución en un sistema empotrado. También comentaremos las herramientas de desarrollo asociadas y veremos como hacer el programa de los LEDs parpadeantes del capitulo anterior. Pero antes de que comencemos, no es sustancialmente diferente de la programación de hemos realizado antes. Lo único que ha cambiado es que cada plataforma de desarrollo es única. Desafortunadamente, esta única diferencia nos añade una complejidad adicional del software, y esta es también la razón por la que debemos poner más atención al proceso de compilación que antes.

## *3.1.- El proceso de producción del software.*

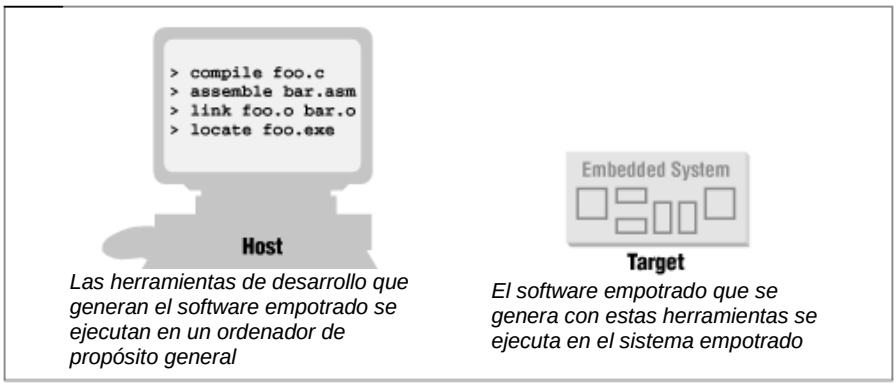
Hay muchas cosas que las herramientas de desarrollo pueden hacer de forma automática cuando la plataforma de desarrollo está bien definida. Esto es posible porque las herramientas conocen las características del hardware y del software donde el programa se va a ejecutar. Por ejemplo, si todos nuestros programas se van a ejecutar en un PC, nuestro compilador puede automatizar y ocultar varios aspectos de la compilación del programa. Las herramientas de desarrollo para software empotrado, por otro lado, raramente hacen suposiciones sobre la plataforma hardware. En vez de esto, el usuario debe proporcionar algo de su conocimiento del sistema a las herramientas dando instrucciones más explicitas.

El proceso de convertir la representación en código fuente del software empotrado en una imagen binaria ejecutable involucra tres pasos distintos. Primero, cada uno de los fichero fuente deben ser compilados o ensamblados en un fichero objeto. Segundo, todos los ficheros objeto que resultan del primer paso deben linkarse juntos para producir un fichero objeto único, denominado programa reubicable. Finalmente, direcciones de memoria física se asignan a los desplazamientos relativos del programa reubicable en un proceso denominado reubicación. El resultado de este tercer paso es un fichero que contiene una imagen binaria ejecutable que está lista para ejecutarse en el sistema empotrado.

El proceso de desarrollo de software empotrado que acabamos de describir se ilustra en la figura siguiente. En esta figura, los tres pasos se han mostrado de arriba abajo, con las herramientas que realizan estos pasos en cajas con los bordes redondeados. Cada una de estas herramientas de desarrollo toma uno o varios ficheros como entrada y produce un fichero único de salida. Información más especifica sobre estas herramientas y los ficheros que producen se encuentran en las secciones que siguen.




Cada uno de los pasos del proceso de desarrollo de software empotrado es una transformación realizada por software que se ejecuta en un ordenador de propósito general (como un PC). Para distinguir entre el ordenador de desarrollo (normalmente un PC o una estación de trabajo UNIX) del sistema empotrado destino, a este se le denomina ordenador Host (huésped). En otras palabras, el compilador, ensamblador y linkador son piezas de software que se ejecutan en un ordenador host, en lugar del sistema empotrado en sí. A pesar del hecho que de se ejecutan en otra plataforma hardware, estas herramientas combinan sus esfuerzos para producir una imagen binaria ejecutable que se ejecute correctamente en el sistema empotrado destino. Esta separación de responsabilidades se muestra en la figura.




En este capitulo y en el siguiente emplearemos las herramientas PICLITE para el sistema basado en PIC y KEIL para el basado en 8051, aunque para sistemas más general se suelen emplearse las herramientas GNU (compilador, ensablador, linkador y debugger).

Estas últimas herramientas son muy populares para los desarrolladores de software empotrado porque se dispone libremente de ellas (no hay que pagar, incluso el código fuente es libre) y soportan la mayoría de los procesadores empotrados.

## *3.2.- Compilación*

El trabajo de un compilador es principalmente traducir los programas escritos en algún lenguaje entendible por los humanos en un juego equivalente de códigos de operación

para un procesador particular. En este sentido, un ensamblador también es un compilador (podríamos llamarlo "compilador de lenguaje ensamblador") pero que realiza una operación más simple de traducción uno a uno de una línea de nemónico legible por los humanos a código de operación equivalente. Todo en esta sección se aplica igualmente a compiladores y a ensambladores. Juntos estas herramientas constituyen el primer paso del proceso de desarrollo de software empotrado.

Por supuesto, cada procesador posee su lenguaje máquina único, por lo que debemos escoger un compilador que sea capaz de producir programas para nuestro procesador destino especifico. En el caso de sistemas empotrados, este compilador casi siempre se ejecuta en el ordenador host. Simplemente no tiene sentido ejecutar el compilador en el propio sistema empotrado. Un compilador como este que se ejecuta en una plataforma y genera código que se ejecuta en otra se llama compilador cruzado. El empleo de un compilador cruzado es una de las características que definen el desarrollo del software empotrado.

El compilador GNU C/C++ (gcc) y ensamblador (as) puede configurarse como compilador nativo o como compilador cruzado. Como compilador cruzado, estas herramientas soportan un juego impresionante de host-destino. La tabla lista los más populares de host y destino soportados. Por supuesto, las selecciones de plataformas host y procesadores destino son independientes, estas herramientas pueden configurarse para cualquier combinación. Además de éstas existen versiones especiales para PIC, HC11 y 8051.

***Hosts y Targets soportados por el compilador GNU***

| Plataformas Host | Procesadores Target |
|---|---|
| | AMD/Intel x86 (32-bit only) |
| DEC Alpha Digital Unix | Fujitsu SPARClite |
| HP 9000/700 HP-UX | Hitachi H8/300, H8/300H, H8/S |
| IBM Power PC AIX | Hitachi SH |
| IBM RS6000 AIX | IBM/Motorola PowerPC |
| SGI Iris IRIX | Intel i960 |
| Sun SPARC Solaris | MIPS R3xxx, R4xx0 |
| Sun SPARC SunOS | Mitsubishi D10V, M32R/D |
| X86 Windows 95/NT | Motorola 68k |
| X86 Red Hat Linux | Sun SPARC, MicroSPARC |
| | Toshiba TX39 |

Independientemente del lenguaje de entrada (C/C++, ensamblador o cualquier otro), la salida del compilador cruzado será un *fichero objeto*. Este es un fichero binario formateado de forma especial que contiene el juego de instrucciones y datos resultado del proceso de traducción del lenguaje. Aunque parte de este fichero contiene código ejecutable, el fichero objeto no está destinado para ejecutarse directamente. De hecho, la estructura interna de un fichero objeto pone de manifiesto que el programa principal no está completo.

El contenido de una fichero objeto puede verse como una estructura de datos flexible y muy grande. La estructura del fichero normalmente viene definida por un estándar como el formato común de ficheros objeto (COFF) o el Formato de Linkador Extendido (ELF). Si estuviésemos empleando más de un compilador (por ejemplo, escribiendo partes de nuestro programa empleando diferentes lenguajes), necesitaríamos asegurarnos de que cada uno es capaz de producir ficheros objeto en el mismo formato. Aunque algunos compiladores (en particular los que funcionan en plataformas Unix) permiten formatos de ficheros objeto como COFF y ELF (gcc soporta ambos), también hay otros que producen ficheros objeto solo en formatos propietarios. Si estamos empleando uno de los compiladores del último grupo, podríamos encontrarnos con que tenemos que comprar todas nuestras otras herramientas de desarrollo al mismo vendedor.

La mayoría de los ficheros objeto comienzan con una cabecera que describe las secciones que siguen. Cada una de estas secciones contiene uno o más bloques de código o datos originarios del fichero fuente original. Sin embargo, estos bloques se han reagrupado por el compilador en secciones relacionadas. Por ejemplo, todos los bloques de código se recogen en una sección denominada text, las variables globales inicializadas (y sus valores iniciales) en una sección denominada data, y las variables globales sin inicializar en una sección denominada bss.

También existe normalmente una tabla de símbolos en algún lado del fichero objeto que contiene los nombres y direcciones de todas las variables y funciones referenciadas por el fichero fuente. Sin embargo, trozos de esta tabla pueden estar incompletos, puesto que no todas las variables y funciones siempre se definen en el mismo fichero. Estos son los símbolos que se refieren a variables y funciones definidas en otros ficheros fuente. Es el linkador el encargado de resolver referencias no resueltas.

## *3.3.- Linkado*

Todos los ficheros objeto resultantes del primer paso deben combinarse de forma especial antes de que el programa pueda ejecutarse. Los ficheros objeto están incompletos de forma individual, la mayoría en que algunas de sus referencias internas a variables y funciones no han sido resueltas todavía. El trabajo del linkador es combinar estos ficheros objeto y, en el proceso, resolver todos los símbolos no resueltos.

La salida del linkador es un fichero objeto nuevo que contiene todo el código y datos de los ficheros objeto de entrada y tiene el mismo formato de ficheros objeto. Esto se realiza mezclando las secciones text, data y bss de los ficheros de entrada. De forma que, cuando el linkador ha finalizado la ejecución, todo el código en lenguaje máquina de todos los ficheros objeto de entrada se encontrarán en la sección text del nuevo

fichero, y todas las variables inicializadas o no residirán en las nuevas secciones data y bss respectivamente.

Además del proceso de unión de los contenidos de las secciones de los ficheros, también realiza la detección de símbolos no resueltos. Por ejemplo, si un fichero objeto contiene una referencia no resuelta a una variable denominada *var* y una variable del mismo nombre se declara en alguno de los otros ficheros objeto, el linkador entenderá que son la misma. La referencia no resuelta será reemplazada con una referencia a la verdadera variable. En otras palabras, si *var* está en la dirección de memoria 14 de la sección data de salida, su entrada en la tabla de símbolos ahora contendrá esta dirección.

El linkador se ejecuta en la misma plataforma que el compilador GNU. Esencialmente es una herramienta de línea de comandos que toma los nombres de los ficheros objeto para linkar como argumentos.

## Código de Arranque (STARTUP)

Una de las cosas que tradicionalmente las herramientas de desarrollo hacen de forma automática es insertar el código de arranque. El código de arranque es un pequeño bloque de código en lenguaje ensamblador que prepara el sistema para la ejecución de software escrito en un lenguaje de alto nivel. Cada lenguaje de alto nivel tiene sus propias expectativas sobre el entorno de ejecución. Por ejemplo, C emplea una pila implícita. Se debe disponer de espacio para la pila y se debe poner a cero antes de que se pueda ejecutar correctamente el software en cualquier lenguaje. Esta es solo una de las responsabilidades asignadas al código de arranque para programas en C.

La mayoría de los compiladores cruzados para sistemas empotrados incluyen un fichero en lenguaje ensamblador denominado *startup.a51, crt0.s* (abrebiatura de C runtime), o algo similar. La colocación y contenido de este fichero se describe normalmente en la documentación facilitada con el compilador.

El código de arranque para programas en C consiste normalmente en las siguientes acciones, realizadas en el orden descrito:

1. Deshabilitar todas las interrupciones.
2. Copiar cualquier dato inicializado de ROM a RAM.
3. Poner a cero la zona de datos no inicializada.
4. Disponer el espacio para la pilar e inicializarla.
5. Inicializar el puntero de pila del procesador.
6. Crear e inicializar el montón (zona de memoria dinámicamente asignable).
7. Habilitar las interrupciones.
8. Realizar la llamada a la función *main.*

Normalmente, el código de arranque también incluye unas pocas instrucciones tras la llamada a main. Estas instrucciones solo serán ejecutadas si el programa de alto nivel sale (es decir, la llamada a main retorne). Dependiendo de la naturaleza del sistema empotrado, podríamos querer usar estas instrucciones para parar el procesador, resetear el sistema, o transferir el control a una herramienta de debugueo.

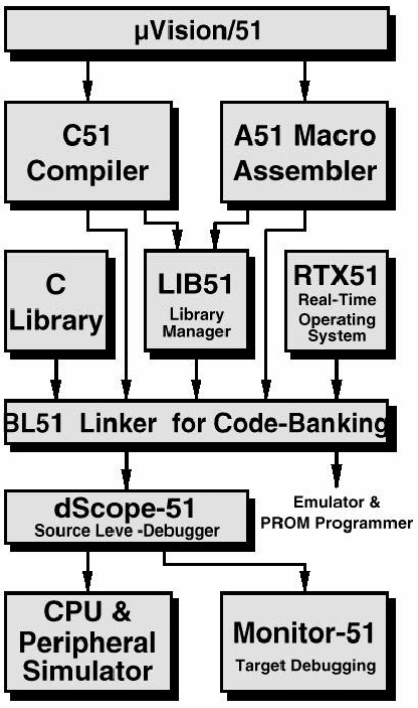
Puesto que el código de arranque generalmente no se inserta de forma automática, el programador debe ensamblarlo él mismo e incluir el fichero objeto resultante junto con la lista de ficheros de entrada al linker. Podría incluso ser necesario ejecutar el linkador con una opción de línea de comando especial para evitar que inserte el código de arranque usual.

Si el mismo símbolo se declara en más de un fichero objeto, el linkador es incapaz de proceder. Normalmente apela al programador mostrando un mensaje de error y saliendo. Sin embargo, si permanece una referencia a un símbolo sin resolver tras la unión de todos los ficheros objeto el linkador tratará de resolver la referencia él mismo. La referencia puede ser a una función que forme parte de la librería estándar, de forma que el linkador abrirá cada una de las librerías descritas en su línea de comando (en el orden establecido) y examinar sus tablas de símbolos. Si encuentra una función con este nombre, la referencia será resuelta incluyendo las secciones code y data asociadas en el fichero objeto de salida.

Desafortunadamente, las rutinas de la librería estándar a veces necesitan algunos cambios antes de que puedan usarse en programación de sistemas empotrados. El problema es que las librerías estándar incluidas en la mayoría de sistemas de desarrollo solo se encuentran en formato objeto. Por lo que raramente tenemos acceso al código fuente de la librería para hacer los cambios necesarios nosotros mismos.

Después de unir todas las secciones code y data y resolver todas las referencias a símbolos, el linkador produce una copia especial “reubicable” del programa. En otras palabras, el programa está completo excepto por una cosa: todavía no se han asignado direcciones de memoria a las secciones code y data. Si no estuviésemos trabajando sobre un sistema empotrado, habríamos terminado de producir nuestro software ahora.

Pero los programadores de sistemas empotrados normalmente no han terminado en este punto el proceso de producción del programa ejecutable. Incluso si nuestro sistema empotrado incluye un sistema operativo, probablemente necesitaremos una imagen binaria del programa ubicada de forma absoluta en memoria.




## *3.4.- Ubicación en Memoria del Programa.*

La herramienta que realiza la conversión del programa reubicable al la imagen binaria ejecutable se denomina “*locator*". Tiene la responsabilidad del paso más simple de los tres. De hecho, tendremos que realizar este paso nosotros mismos muchas veces, aportando información sobre la memoria de la placa de desarrollo como entrada al programa. El locator empleará esta información para asignar direcciones de memoria física a cada una de las secciones code y data.

En algunos casos, el locutor es una herramienta de desarrollo separada. Sin embargo, en el caso del entorno KEIL, esta funcionalidad está incluida en el linkador BL51. No hay que confundirse con esta implementación particular. Cuando escribimos código para un computador de propósito general o un sistema empotrado, en algún momento a nuestro programa reubicable se le tendrán que asignar direcciones reales. Esto es cierto incluso si el locator es parte del linkador, como es el caso de BL51.

La organización de memoria necesaria por la herramienta se le puede pasar mediante un fichero de configuración o directamente en el entorno en la ventana izquierda donde fijamos las opciones de enlazado de nuestro proyecto (en la ventana de uvision). En la siguiente figura se observa el proceso de generación de programas en el entorno uVision de KEIL.

El resultado de este paso final es una imagen binaria dispuesta en memoria de forma absoluta que puede descargarse en el sistema empotrado o programarse en un dispositivo de memoria de solo lectura.

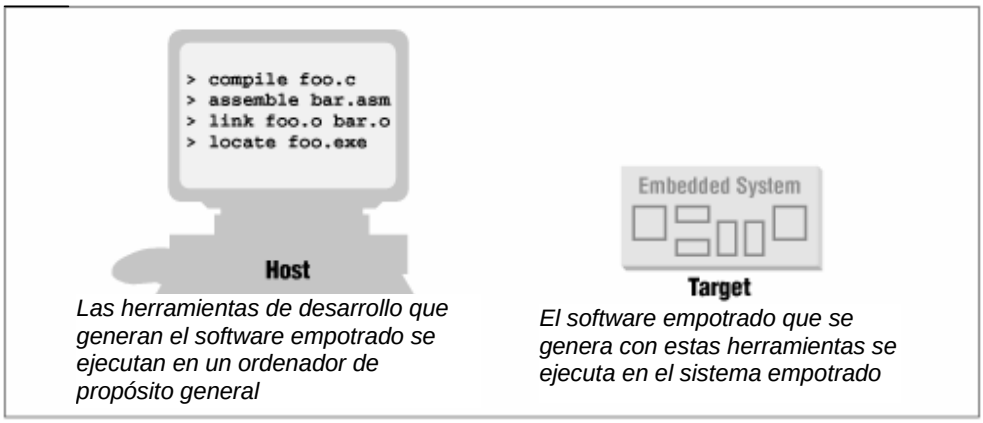
En el ejemplo anterior del LED, el programa BL51 genera un listado con información de los módulos que se han compilado, las variables globales. En la siguiente tabla podemos observar el mapa de memoria de la última fase de desarrollo del programa.

```
LINK MAP OF MODULE:  pru (GENERADOR)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            IDATA   0008H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     000CH     UNIT         ?C_C51STARTUP
            CODE    000FH     0009H     UNIT         ?PR?MAIN?GENERADOR
            CODE    0018H     0007H     UNIT         ?PR?_RETARDO?GENERADOR
```

# 4.- Descarga y depuración.

Una vez que disponemos de la ***imagen binaria*** almacenada como un fichero en el ordenador de desarrollo (archivo generalmente con la extensión BIN) o almacenada con formato INTEL HEX (archivo ascii con extensión HEX), necesitamos alguna manera de descargar esta imagen al sistema empotrado y ejecutarla. La imagen binaria normalmente se carga en un dispositivo de memoria sobre la tarjeta de desarrollo y se ejecuta desde esta. Si disponemos de las herramientas adecuadas, se podrán colocar puntos de parada en el programa o bien observar su ejecución de la manera menos intrusiva posible. Este capitulo describe varias técnicas para descargar, ejecutar y depurar software empotrado.

Lo primero es diferenciar entre el sistema de desarrollo y el sistema destino o "target". El sistema de desarrollo o "host" suele ser un PC que ejecuta un compilador cruzado que genera código binario para un determinado tipo de procesador. El programa se prueba en una placa de desarrollo o en el prototipo final.




*Las herramientas de desarrollo que generan el software empotrado se ejecutan en un ordenador de propósito general*

*El software empotrado que se genera con estas herramientas se ejecuta en el sistema empotrado*

## *4.1.- Cuando lo hacemos en ROM…*

Una de las formas más obvias de descargar el software es cargar la imagen binaria en un dispositivo de memoria de solo lectura e insertar este chip en el zócalo de la placa de desarrollo. Claro está, el contenido de una verdadera memoria de solo lectura no se puede sobrescribir. Sin embargo, los sistemas empotrados emplean dispositivos de solo lectura especiales que pueden programarse o reprogramarse con la ayuda de un dispositivo especial denominado ***programador de dispositivos***.

Un programador de dispositivos es un sistema computador que dispone de varios zócalos de memoria de varias formas y tamaños y es capaz de programar dispositivos de memoria de varios tipos.

En un escenario de trabajo ideal, el programador se conecta a la misma red del ordenador de desarrollo. De esta forma los ficheros que contienen binarios ejecutables pueden transferirse fácilmente para programación de ROM. Cuando la imagen binaria se ha transferido al programador, el chip de memoria se coloca en el zócalo apropiado y se selecciona el tipo de dispositivo desde el menú en pantalla. El proceso de programación puede tomar desde unos pocos segundos a varios minutos, dependiendo del tamaño de la imagen binaria y del tipo de memoria que se esté usando.

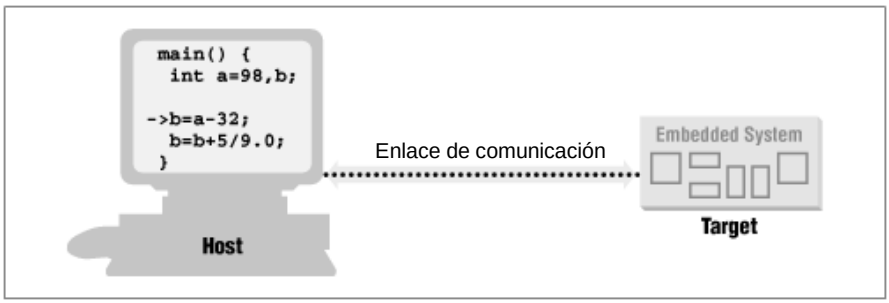
Tras la programación de la ROM, esta se encuentra lista para insertarse en su zócalo en la placa. Por supuesto, esto debe hacerse con el sistema empotrado desconectado. Tan pronto como se activa la corriente de la placa el procesador siguiendo su propia regla busca la primera instrucción a ejecutar (el 8051 en la dirección 0000H). Esto se denomina dirección de reset.

La mayor desventaja de esta técnica de descarga es que no hay forma sencilla de probar el software que se ejecuta desde ROM. El procesador busca y ejecuta las instrucciones tan rápido como puede y no proporciona ninguna forma de observar el estado interno del programa. Esto puede ser útil una vez que sabemos que nuestro programa funciona y estamos listos para fabricar en serie el sistema, pero no resulta útil durante el desarrollo del sofware. Por supuesto, podemos examinar el estado de los LEDs y otro hardware visible externamente pero esto nunca nos proporcionara tanta información como un depurador ( “debugger”).

## *4.3.- Depuradores remotos o monitores.*

Si lo tenemos disponible, un depurador remoto puede emplearse para descargar, ejecutar y probar software empotrado sobre un puerto serie o conexión de red entre el sistema de desarrollo y el sistema empotrado. El entorno de trabajo es como el de cualquier sistema de prueba ya empleado. Sin embargo, en el caso de sistemas empotrados, el depurador y el software que se esta probando se ejecutan en dos procesadores diferentes.

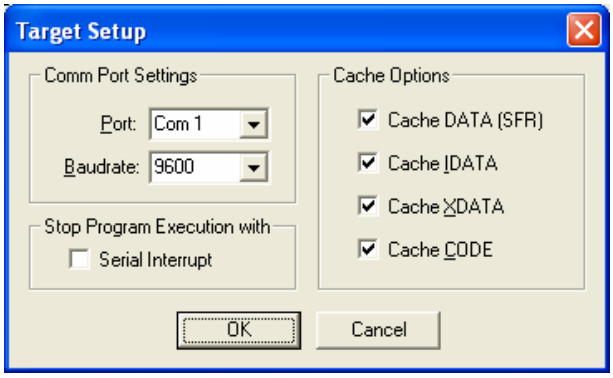
Un depurador remoto consiste en dos piezas de software, una es la interfaz de usuario que se ejecuta en el host y que permite interaccionar con el programa que se está probando. La otra es un pequeño programa que se ejecuta en el sistema empotrado (target) y que proporciona la capacidad de carga del programa, ejecución, parada y volcado de toda la información del entorno de ejecución que nos permite depurar el programa.




El monitor de depuración reside en memoria ROM y ha sido colocado de la manera comentada anteriormente. El sistema de desarrollo debe tener suficiente memoria RAM desde la cual pueda guardarse y ejecutarse el programa que estamos desarrollando.

Por ejemplo, el monitor PAULMON es de libre distribución y funciona en el 8051 y permite cargar y depurar el programa.

El entorno de programación para 8051 microvision de KEIL permite la conexión vía puerto serie RS-232 de un monitor de depuración.




## *4.4.- Emuladores en circuito.*

Los emuladores remotos son útiles para monitorizar y controlar el estado del software empotrado, pero solo un emulador en circuito (ICE) nos permite examinar el estado del procesador en el que se ejecuta el programa. De hecho, un ICE realmente sustituye o emula el procesador de la tarjeta. Es en sí mismo un sistema empotrado, con su propia copia del procesador destino, RAM, ROM y su propio software empotrado. Como resultado, los emuladotes en circuito son generalmente bastante caros, mucho más que el hardware empotrado destino. Sin embargo constituyen una herramienta potente, y en casos complicados ninguna otra herramienta ayudará a realizar el trabajo mejor.

Como un monitor de depuración, un emulador emplea un depurador remoto para su interfaz de usuario. Como el emulador tiene su propia copia del procesador destino es posible ver y controlar el estado del procesador en tiempo real. Esto permite al emulador dar soporte a potentes características de depuración como puntos de parada especiales y traceado en tiempo real, además de las características proporcionadas por cualquier monitor de depuración.


Con un monitor, podemos colocar puntos de parada en el programa, pero estos están limitados a búsquedas de instrucción, o sea que el procesador se para cuando va a buscar una determinada instrucción. Los Emuladores por el contrario, también soportan puntos de parada provocados por el hardware. Estos nos permiten parar la ejecución en respuesta a varios tipos de eventos, no solo búsquedas de instrucción, si no lecturas y escrituras de E/S y de memoria e interrupciones.

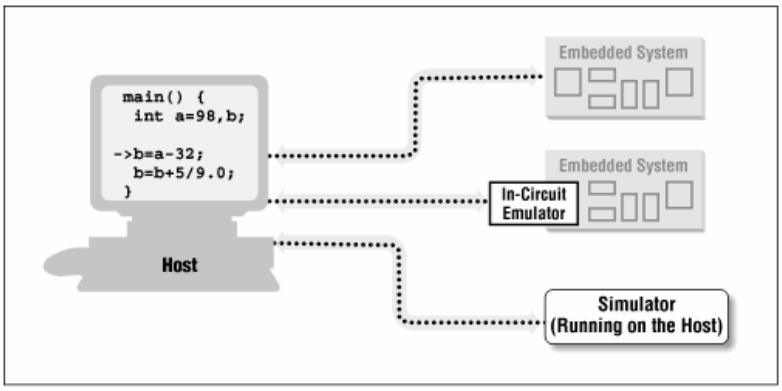
El traceado en tiempo real consiste en obtener información sobre los tipos de ciclo que ha ejecutado el procesador y cuanto tiempo ha perdido en la ejecución de cada función de nuestro programa en lenguaje C, por ejemplo. Esta información es muy útil a la hora de optimizar el programa o encontrar fallos.

Otro tipo de emulador diferente es un ***emulador de ROM***. Es un dispositivo que emula un dispositivo de memoria de solo lectura. Como un ICE, es un sistema empotrado que se conecta al sistema destino y comunica con el Host. Sin embargo, esta vez la conexión con el sistema destino se hace a través del zocalo de ROM. Para el procesador empotrado, es como otro dispositivo de memoria. Pero para el depurador, funciona como un monitor de depuración.

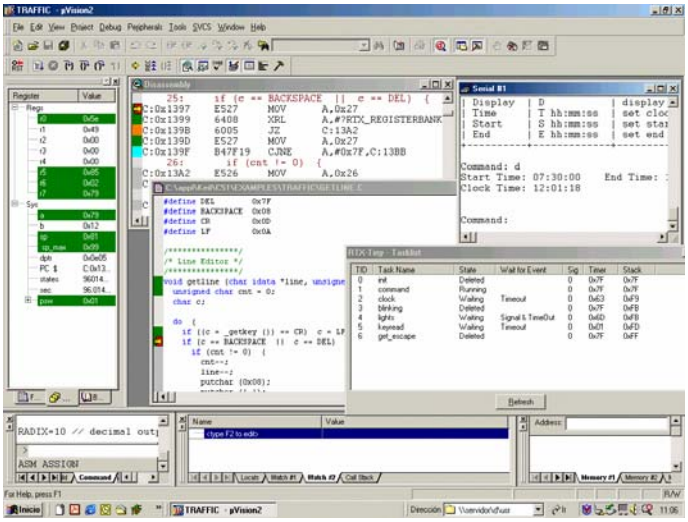
Tienen algunas ventajas sobre otros monitores. Primero, no es necesario programar un programa monitor adecuado para el hardware. Segundo, el emulador de ROM dispone de su propia conexión con el Host, por lo que no resulta necesario emplear la conexión del sistema empotrado. Y finalmente, el emulador de ROM es un sustituto de la ROM original, por lo que no se emplea la memoria del sistema empotrado para almacenar el código del monitor.

## *4.5.- Simuladores y otras herramientas.*

Hay disponibles varias herramientas de depuración, incluyendo simuladores, analizadores lógicos y osciloscopios. Un simulador es un programa ejecutado en el Host que simula la funcionalidad y el juego de instrucciones del procesador destino. La interfaz de usuario normalmente es la misma o similar a la de un depurador remoto. Aunque los simuladores tienen varias desventajas, son bastante útiles en las primeras etapas de un proyecto cuando todavía no tenemos hardware para que el programador pueda experimentar con él.

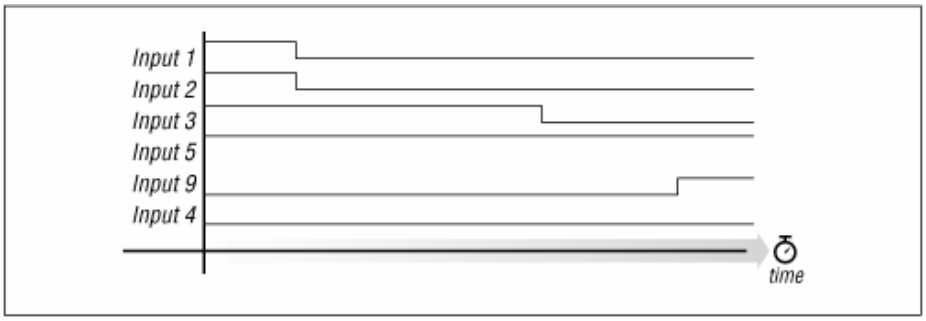



La principal desventaja de un simulador es que solamente simula el procesador. Un sistema empotrado generalmente contiene varios periféricos importantes. La interacción con estos dispositivos a veces es imitada por el simulador (por ejemplo, el puerto serie, los timers, etc), pero estos a veces pueden crear problemas.



Una vez que tengamos acceso al sistema de prueba y especialmente durante la depuración del hardware, los analizadores lógicos y osciloscopios resultan herramientas indispensables. Principalmente son útiles para depurar las interacciones entre el procesador y otros chips de la placa. Puesto que no pueden ver señales internas del procesador, no pueden controlar el flujo de ejecución del software como puede hacer un depurador o un emulador. Por lo que solos por si mismos no ofrecen mucha utilidad, pero controlados por un depurador remoto o un emulador en circuito pueden ser extremadamente útiles.

Un *analizador lógico* es un equipo de laboratorio diseñado para reparaciones de hardware digital. Puede tener muchas entradas, cada una capaz de detectar solo una cosa: si la señal lógica conectada tiene nivel 1 o nivel 0. Cualquier subconjunto de entradas que seleccionemos pueden ser mostradas en la pantalla frente al tiempo. La mayoría de analizadores permiten capturar los datos como respuesta a un patrón particular.




# 5.- Metodología para conocer el hardware.

## *5.1.- Análisis de Sistemas.*

Antes de escribir el software de un sistema empotrado, debemos familiarizarnos con el hardware en el que se va a ejecutar. Al principio debemos conocer solo el funcionamiento general del sistema.

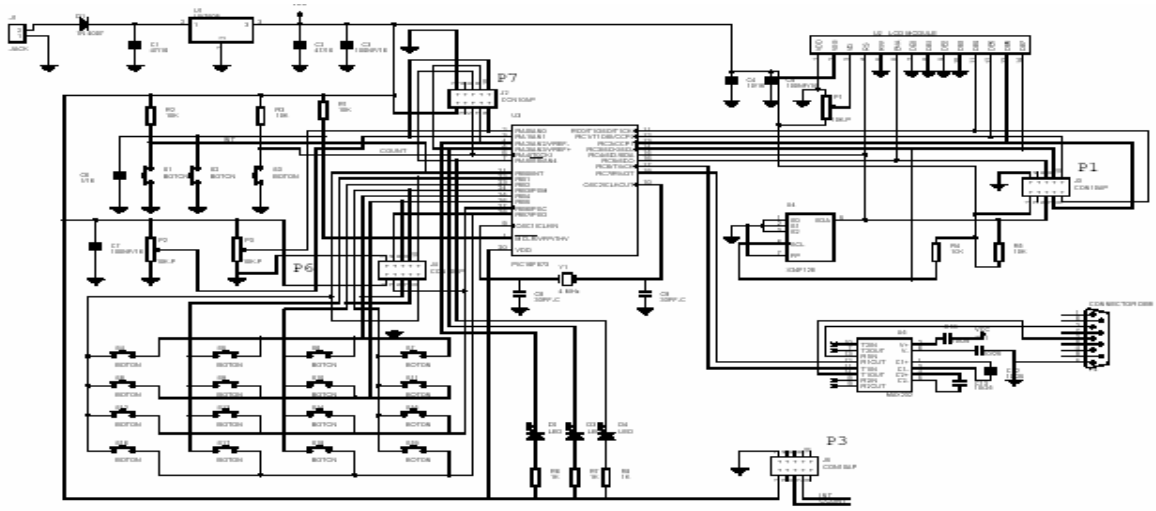
### 5.1.1.- Estudio del esquema eléctrico.

Si el sistema que vamos a programar es un producto terminado debe contener un manual de algún tipo, puede ser una guía de usuario o un manual del programador. Este ha sido escrito para el desarrollador de software en mente. Pero si la placa se ha diseñado a medida para el proyecto, la documentación puede ser más escasa y compleja y pensada como referencia. En cualquier caso, este es el mejor sitio por donde empezar.

A medida que estudiamos la documentación nos debemos hacer las siguientes preguntas:

- ¿Cuál es el propósito del sistema?
- ¿Cuáles son sus principales componentes y que función realiza cada uno?
- ¿Como circulan los datos que procesa el sistema a través de estos componentes?

Vamos a suponer que recibimos una tarjeta de la cual tan solo tenemos el esquema. Si nos encargamos de programar el software deberemos contestarnos a estas preguntas.




En el esquema de la figura superior, podemos identificar varios componentes principales:

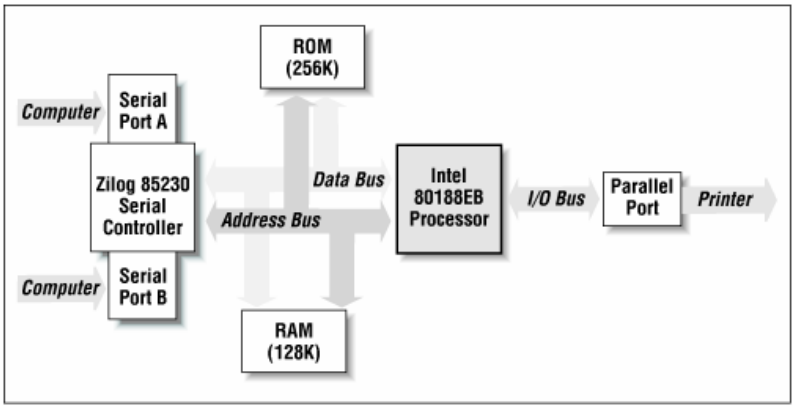
- **Teclado**: esto será una entrada de datos.
- **Puerto Serie**: entrada/salida de datos.
- **Fuente de Alimentación**: significa que no necesita fuente separada.
- **Conectores de Expansión**: significa que se pueden conectar algo externo.
- **Microcontrolador**: dispositivo a controlar.
- **Display LCD**: salida de datos.
- **LEDs**: salida de datos.
- **Memoria EEPROM**: almacenaje de datos externo al microprocesador.
- **Amplificador de niveles lógicos**: habrá que ver porqué esta aquí.

Una vez identificados los componentes principales, deberemos obtener toda la información que podamos de cada uno. Si se trata de componentes electrónicos, podemos conseguir las Hojas de características (*data sheets*) de la biblioteca o de Internet y con estas conocer la función y la forma de programar y controlar estos componentes periféricos.

Si conocemos el objetivo final del sistema, podemos imaginar como circula el *flujo de datos*. Si el sistema es un control de acceso, podemos suponer que la información se obtiene del teclado y se muestra en la pantalla LCD, esta información puede controlar un relé mediante los puertos de salida y ser enviada por la conexión serie RS-232 a otro lugar para ser procesada.

## 5.1.2.- Diagrama de Bloques.

Otro ejemplo puede ser un dispositivo para compartir una impresora. El diagrama siguiente muestra el sistema. Analizando el *diagrama de bloques* debemos ser capaces de visualizar rápidamente el flujo de datos a través del sistema.




Podemos observar que las dos fuentes de datos se conectan a través de sendos puertos serie incluidos en el controlador serie 85230. Ni que decir tiene que el data sheet de este circuito integrado nos aportaría mucha información de la programación del puerto serie. El procesador transfiere la información de los puertos serie a un puerto paralelo que está conectado a la impresora, decidiendo en que orden se entregan los datos procedentes de los PCs.

Una vez realizado el *diagrama de bloques*, podemos incluirlo con la documentación del proyecto. De esta forma a medida que continuemos trabajando en el proyecto vamos aumentando el contenido de información que posteriormente podemos emplear para utilizarla como referencia para nosotros en el futuro o para otros que trabajen con el sistema.

También resulta útil ponerse en el papel del microprocesador del sistema, de hecho, somos nosotros los que lo vamos a programar y debemos pensar y actuar como lo hace el microcontrolador, sobre todo cuando se relaciona con los demás periféricos internos o externos.

## *5.2.- Diseño de sistemas.*

El diseño de sistemas, frente al análisis, requiere la respuesta a las mismas preguntas anteriores, pero esta vez desde otro punto de vista. Muchas veces partimos de un sistema ya fabricado o modificamos uno que funciona en base a alguna necesidad de diseño particular.

Ahora la pregunta más importante es *la finalidad del sistema*. Por ejemplo, todos sabemos que la finalidad de un reproductor de CDs es la reproducción de música almacenada en un CD. Sin embargo, el funcionamiento concreto de la interfaz con el usuario, el orden de reproducción y otros detalles han sido escogidos por el diseñador del hardware y del software.

Lo mismo sucede con el diseño del software de un sistema empotrado, los detalles los tenemos que completar nosotros en función de la finalidad del sistema y de las restricciones del hardware, como por ejemplo, la pantalla y pulsadores empleados.

Debemos contestar la siguiente pregunta: *¿Cuál es la finalidad del sistema?*. Para acercarnos a todos los detalles resulta conveniente seguir los siguientes pasos:

1. Hacer una **lista de funciones** que realiza y cuales son importantes y cuales superfluas.
2. **Ordenar** las funciones en el tiempo.
3. Realizar **diagrama de flujo** a partir de estas funciones. Empezar con el diagrama de flujo básico y luego irlo detallando.

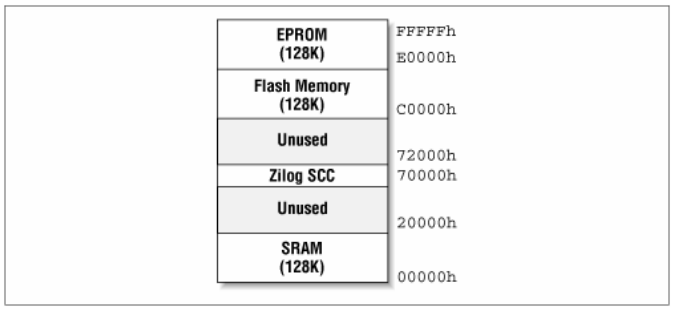
La otra pregunta importante en sistemas de procesamiento de datos es la siguiente: *¿Cuál es el flujo de datos?*. Para ayudar en la confección de la parte del programa que se va a encargar de los datos resulta conveniente proceder de la forma siguiente:

1. Realizar una **lista de datos** que procesa
2. Realizar una **lista de variables** (arrays) importantes y no repetir de forma innecesaria variables compartidas por varias tareas.

## *5.2.- Mapa de Memoria.*

Todos los procesadores almacenan los programas y datos en memoria. En algunos casos esta memoria reside en el interior del microcontrolador, pero otras está en componentes externos. Estos componentes se ubican en el espacio de memoria del microprocesador, y el procesador se comunica con ellos mediante cables eléctricos denominados *bus de direcciones y bus de datos*.

Un mapa de memoria es una representación gráfica de la distribución de memoria y periféricos en todo el rango direccionable mediante el bus de direcciones interno o externo del microprocesador.




En el diagrama anterior se puede observar el mapa de memoria del sistema anterior basado en el 80C188 que tiene un rango de 0xFFFFF o 1Mbyte. Dentro de este rango se localizan 128Kbytes de memoria SRAM (RAM estática) a partir de la dirección 0, el periférico serie Zilog SCC a partir de la dirección 0x70000, la memoria FLASH (EEPROM) de 128Kbytes a partir de la dirección 0xC0000 y por último la EPROM de 128K que almacena el programa monitor a partir de la dirección 0xE0000.

Si queremos programar el periférico serie, tendremos que escribir a partir de la dirección de memoria 0x70000 para controlar el envío/recepción de datos a través del puerto serie. Esta dirección se denomina *dirección base* del periférico.

Hay que tener en cuenta que a partir del esquema eléctrico se deduce mapa de memoria del sistema. Para ello se emplea un circuito combinacional decodificador de direcciones que en el rango asignado selecciona el chip activando una determinada entrada que suele tener el nombre CS de *chip select*.

Por tanto, conocer el mapa de memoria resulta imprescindible para programar el sistema y para realizar ampliaciones y modificaciones.

En algunos sistemas que no disponen de memoria externa tienen mapa predeterminado por fabricante, por ejemplo, el 8051 dispone 128 bytes en un *área de registros de función especial (SFR)* definidos por el fabricante.

En cuanto a la programación de estos periféricos mapeados en memoria, normalmente la dirección de memoria donde se encuentran se coloca en un fichero de cabecera y se tiene también para documentación. Esta dirección suele llamarse *dirección base* porque a partir de esta se calculan las direcciones de cada registro de control o de estado del periférico. Es mediante estos registros como se programa y se cambia la funcionalidad del periférico.

```
/*********************************************************************
 *
 *
 *  I/O Map
 *
 *            Base Address    Description
 *            --------------- ---------------------------------------
 *            0000h           Unused
 *            FC00h           SourceVIEW Debugger Port (SVIEW)
 *            FD00h           Parallel I/O Port (PIO)
 *            FE00h           Unused
 *            FF00h           Peripheral Control Block (PCB)
 *

*********************************************************************
/

#define         SVIEW_BASE   0xFC00
#define         PIO_BASE     0xFD00
#define        PCB_BASE      0xFF00
```

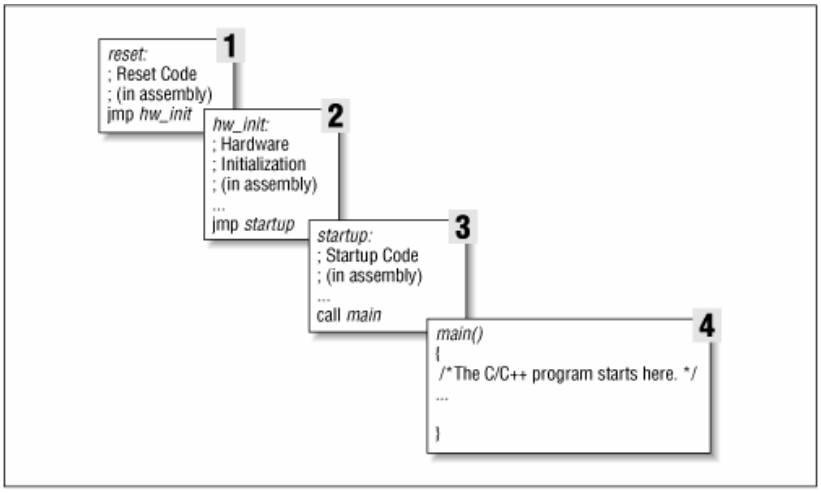
## *5.3.- Inicialización de Software y Hardware.*

El último paso para conocer nuestro hardware es escribir algún software para su inicialización. Es la mejor oportunidad para conocer de forma estrecha las relaciones entre el software y el hardware, sobre todo si vamos a emplear después un lenguaje de alto nivel, como es el lenguaje ‘C’.

Todos los sistemas de desarrollo necesitan una cierta cantidad de *software de testeo* para asegurar que el hardware está correctamente diseñado y que los periféricos funcionan correctamente. Cuando algo no funciona correctamente nos encontramos en una posición difícil, ¿Qué es lo que está fallando, el software o el hardware?

Si tenemos acceso a un simulador podemos **construir algunas pruebas o experimentos** para contestar a esta pregunta, en otro caso deberemos trabajar en el laboratorio directamente con el prototipo.

Al emplear un compilador de C para microcontroladores, la **inicialización hardware básica** tiene lugar antes de ejecutar main(). Esta inicialización depende del microprocesador empleado y del fabricante del compilador y se suele realizar en ensamblador en el fichero *STARTUP.ASM o crt.s.* Esta inicialización es muy básica y no incluye la inicialización de los periféricos que no sean básicos ni de las interrupciones. La inicialización de los periféricos empleados la deberemos realizar en nuestro programa antes de emplear los periféricos afectados.



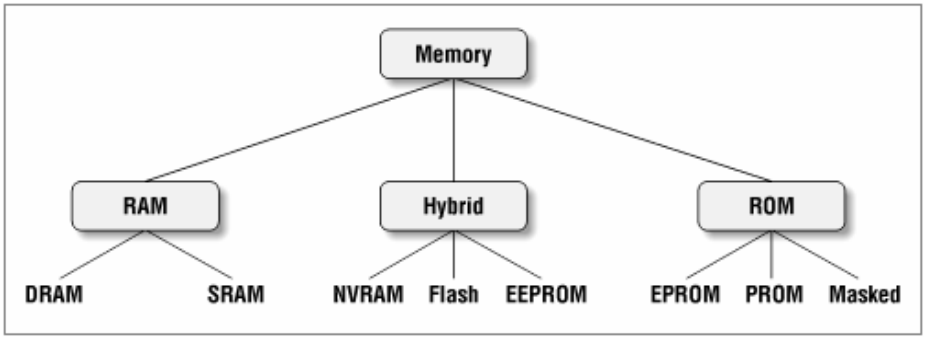

A continuación se realiza la **inicialización software**. Esta puede incluir la copia de las variables inicializadas en el programa fuente 'C', la puesta a cero del espacio de memoria empleado, etc. Luego se llama a main(), con lo que **comienza** nuestro programa.

Dentro de main se inicializan los periféricos que usa nuestro programa, incluyendo timers e interrupciones si se emplean.

## *5.4.- Tipos de Memoria del Sistema.*

Existen muchos tipos de memoria en los sistemas que podemos programar, pero básicamente estos tipos se pueden dividir en los siguientes tres:

- **RAM:** SRAM o DRAM: de tipo volátil, almacena las variables del programa.
- **HIBRIDA:** EEPROM, FLASH, NVRAM: Almacena una pequeña cantidad de datos de configuración o bien el programa completo.
- **ROM:** Máscara, PROM o EPROM: Almacena el programa principal o el sistema básico de carga del programa.




En el caso de nuestro sistema de desarrollo del curso, la placa II2, dispone de varios tipos de memoria:

- **Memoria de datos interna al procesador**, es una memoria RAM donde se almacenan los registros del procesador y las variables del programa.

- **Memoria de programa interna (FLASH)**, esta memoria almacena el programa principal y un pequeño programa denominado "cargador" que tiene la finalidad de obtener el programa principal a través del puerto serie y grabarlo en la memoria de programa. Esta memoria se puede borrar de forma eléctrica, con lo que no es necesario un programador para el desarrollo del sistema.

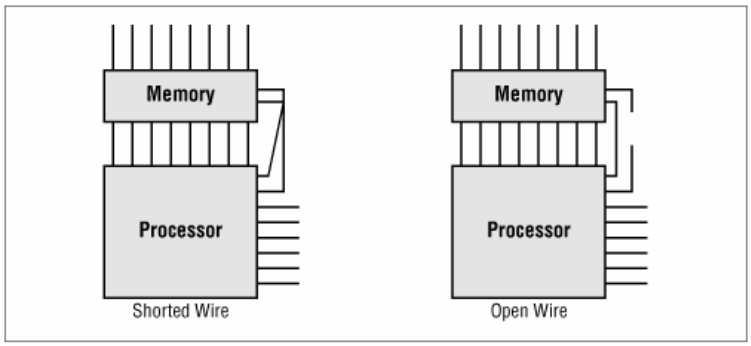
- **Memoria EEPROM interna**, esta memoria también es borrable de forma eléctrica, pero tiene una capacidad mucho menor que la anterior. Su principal función el guardar pequeños datos de configuración empleados por el programa.

- **Memoria EEPROM externa**, esta memoria también es borrable de forma eléctrica y tiene una capacidad mayor que la EEPROM interna, se emplea para guardar datos de configuración que ocupen mayor espacio.

## *5.5.- Técnicas de testeo de memoria.*

La técnicas de testeo de memoria se emplean en sistema microcontroladores que disponen de memoria externa y tienen la finalidad de *detectar problemas de montaje y fabricación*. Estos problemas pueden ocasionar fallos de acceso a memoria e incluso que no arranque el sistema. Pueden darse los siguientes:

- Problemas con el cableado entre el procesador y la memoria.
- Chips de memoria no colocados.
- Chips de memoria mal insertados.




Para detectar estos fallos se emplean varias estrategias de test, entre las cuales las más empleadas son las siguientes:

- **Testeo del bus de datos** con un ¨1¨ desplazándose entre los bits: de esta forma se pueden detectar líneas en cortocircuito o abiertas.
- **Testeo del bus de direcciones** con un "1" desplazándose entre los bits: si se repiten patrones de datos escritos en la dirección correcta será porque la línea de direcciones está abierta o está cortocircuitada.

- Testeo de la memoria en sí usando un **patrón incremental**: puede ser necesario para distinguir posiciones anteriormente escritas y/o leidas.

# 6.- Manejo de periféricos.

Cada periférico interno o externo que se encuentre mapeado en memoria se accede a través de uno o varios registros de control y registros de estado. Estos registros permiten la programación del modo de funcionamiento, el envío de órdenes y la recepción de datos o del estado del periférico.

## *6.1.- Registros de control y de estado.*

Los *registros de control* son registros de solo escritura donde se programan las funciones u órdenes para el periférico. Los *registros de estado* son registros de solo lectura donde se lee el estado del periférico. En principio estos registros pueden estar combinados en la misma dirección de memoria o no, puesto que la operación de lectura y la de escritura son operaciones diferentes que tienen significados diferentes y por tanto los registros son diferentes aunque se encuentren en la misma dirección de memoria.

No hay *forma estándar de acceder en C los periféricos*, por lo que o bien se accede en ensamblador o se accede a través de punteros. Estos punteros coinciden con la dirección de memoria donde se encuentran los periféricos. Si estos se encuentran en espacios de direcciones especiales, como sucede en el PIC o en el 8051, el compilador integra un mecanismo especial para permitir este acceso. Este mecanismo depende del compilador, por lo que hay que estudiarse el manual de usuario del compilador para averiguar como se puede acceder a los periféricos.

En capítulos posteriores veremos como se realiza este acceso en el 8051, compilador KEIL y en el PIC con el compilador HI-TECH.

## *6.2.- Controlador de dispositivo (device driver).*

Un *Device Driver* es un módulo software que contiene todas las instrucciones en C o en ensamblador necesarias para programar un determinado dispositivo. Esta forma de hacer las cosas tiene la ventaja de que oculta al programador la complejidad de la programación de bajo nivel o básica.

Por ejemplo, si necesitamos imprimir un carácter en el display LCD que integra la placa de desarrollo II2 podemos hacerlo de dos formas:

1. **Podemos consultar el data sheet** del LCD y el esquema eléctrico para ver como está conectado al microcontrolador y programar el microcontrolador para realizar todas las funciones de inicialización que indica el fabricante del LCD. Una vez realizadas las funciones de inicialización a través del puerto del microcontrolador, debemos programar el LCD para que imprima el carácter.

2. **Podemos emplear una librería** o *device driver* de manejo del LCD realizado por una tercera persona que conoce el LCD y la placa de desarrollo. Este driver se emplea incluyendo en el código fuente de mi programa un fichero de cabecera que lista los prototipos de las funciones que puedo emplear. Junto con los prototipos se incluye una breve descripción de la finalidad de cada función disponible. Hay una función denominada *lcd_init()* que inicializa el LCD como indica el fabricante y otra denominada *lcd_putch()* que toma como parámetro un char y lo imprime en la posición del cursor. Pues para hacer mi programa tan solo tendré que llamar a estas dos funciones secuencialmente.

El segundo enfoque hace más rápido el desarrollo del software y permite reutilizar módulos nuevos que se vayan haciendo, además de resultar más sencillo para el programador ocasional de la placa de desarrollo.

Este será el enfoque empleado a lo largo del curso en la asignatura y en las prácticas.

## *6.3.- Las librerías de periféricos.*

Para realizar librerías de periféricos en C se realizan unos pasos muy sencillos. Todo depende del tipo de periférico y de su complejidad.

Se suele comenzar creando una *estructura de datos o un puntero* que mapee en memoria el conjunto de registros de lectura y escritura del periférico. Si se trata de un periférico interno, estos registros ya pueden venir especificados por el fabricante del compilador, como sucede con el PIC, que tiene todos los registros listados en el fichero de cabecera *pic.h*.

Por ejemplo en el siguiente listado se ha empleado una estructura de lenguaje 'C' para contener los registros básicos para el manejo de un temporizador/contador, además de definir ciertas constantes útiles.

```
struct TimerCounter
{
    unsigned short  count;              // Current Count, offset 0x00
    unsigned short  maxCountA;          // Maximum Count, offset 0x02
    unsigned short  _reserved;          // Unused Space,  offset 0x04
    unsigned short  control;            // Control Bits,  offset 0x06
};

#define TIMER_ENABLE    0xC000          // Enable the timer.
#define TIMER_DISABLE   0x4000          // Disable the timer.
#define TIMER_INTERRUPT 0x2000          // Enable timer interrupts.
#define TIMER_MAXCOUNT  0x0020          // Timer complete?
#define TIMER_PERIODIC  0x0001          // Periodic timer?
```

Una vez definidas las constantes, se hace necesario *definir un juego de variables* para mantener el estado del dispositivo. Si el dispositivo no tiene estado esto no es necesario.

Habrá que crear *funciones para inicializar el periférico*. Estas funciones se deberán ejecutar la primera vez que se intente usar el periferico y generalmente se encontraran en una función de inicialización que se ejecute antes del resto del programa y fuera del bucle principal del programa.

Por supuesto se deben crear *funciones* que en conjunto proporcionen una interfaz a los usuarios del device driver para poder *controlar el dispositivo en su totalidad*. Si esto no es posible siempre se puede dejar una función para permitir un control de más bajo nivel del periférico si esto fuese necesario.

Si el periférico se controla mediante interrupciones será necesario *crear las RSI* (rutinas de servicio de interrupción) que coordinen el funcionamiento del periférico.