Escuela Politécnica Superior
Universidad de Huelva

Departamento de Ing. Electrónica,
Sistemas Informáticos y Automática

TERCER CURSO. INFORMÁTICA INDUSTRIAL II

CREACIÓN DE PROGRAMAS EN C PARA LA FAMILIA MCS-51

Manuel Sánchez Raya
Versión 0.99
30 de Noviembre de 2002

ÍNDICE

BIBLIOGRAFÍA:

Texto original en inglés: **Mike Beach**, Hitex (UK) Ltd

Traducción al castellano por Sergio Aurtenetxea, s_aurtenetxea@hispavista.com,
Universidad del Pais Vasco.

Capitulo 1. Introducción.

C es un lenguaje bastante conciso y en ocasiones desconcertante. Considerado ampliamente como un lenguaje de alto nivel, posee muchas características importantes, tales como: programación estructurada, un método definido para llamada a funciones y para paso de parámetros, potentes estructuras de control, etc.

Sin embargo gran parte de la potencia de C reside en su habilidad para combinar comandos simples de bajo nivel, en complicadas funciones de alto nivel, y en permitir el acceso a los bytes y words del procesador. En cierto modo, C puede considerarse como una clase de lenguaje ensamblador universal. La mayor parte de los programadores familiarizados con C, lo han utilizado para programar grandes máquinas que corren Unix o últimamente MS-DOS. En estas máquinas el tamaño del programa no es importante, y el interface con el mundo real se realiza a través de llamadas a funciones o mediante interrupciones DOS. Así el programador en C sólo debe preocuparse en la manipulación de variables, cadenas, matrices, etc.

Con los modernos microcontroladores de 8 bits, la situación es algo distinta. Tomando como ejemplo el 8051, el tamaño total del programa debe ser inferior a los 4 u 8K, y debe usarse menos de 128 o 256 bytes de RAM. Idealmente, los dispositivos reales y los registros de funciones especiales deben ser direccionados desde C. Las interrupciones, que requieren vectores en direcciones absolutas también deben ser atendidas desde C. Además, se debe tener un cuidado especial con las rutinas de ubicación de datos para evitar la sobre-escritura de datos existentes.

Uno de los fundamentos de C es que los parámetros (variables de entrada) se pasan a las funciones (subrutinas) en la pila, y los resultados se devuelven también en la pila. Así las funciones pueden ser llamadas desde las interrupciones y desde el programa principal sin temor a que las variables locales sean sobre-escritas (re-entrancia).

Una seria restricción de la familia 8051 es la carencia de una verdadera pila. En un procesador como el 8086, el apuntador de la pila tiene al menos 16 bits. Además del apuntador de pila, hay otros registros que pueden actuar como apuntadores a datos en la pila, tal como el BP (*Base Pointer*). En C, la habilidad para acceder a los datos en la pila es crucial. Como ya ha sido indicado, la familia 8051 está dotada de una pila que realmente sólo es capaz de manejar direcciones de retorno. Con 256 bytes disponibles, como máximo, para la pila no se pueden pasar muchos parámetros y realizar llamadas a muchas funciones.

De todo ello, puede pensarse que la implementación de un lenguaje que como C haga un uso intensivo de la pila, es imposible en un 8051. Hasta hace poco así ha sido. El 8051, hace tiempo que dispone de compiladores C, que en su mayor parte han sido adaptados de micros más potentes, tal como el 68000. Por ello la aproximación al problema de la pila se ha realizado creando pilas artificiales por software. Típicamente se ha apartado un área de RAM externa para que funcione como una pila, con la ayuda de rutinas que manejan la pila cada vez que se realizan llamadas a funciones. Este método funciona y proporciona capacidad de re-entrancia, pero a costa de hacer los programas muy lentos. Por lo tanto, con la familia 8051, la programación en lenguaje ensamblador ha sido la única alternativa real para el desarrollo de pequeños sistemas en los que el tiempo es un factor crítico.

Sin embargo, en 1980, Intel proporcionó una solución parcial al problema al permitir la programación del 8051 en un lenguaje de alto nivel llamado PLM51. Este compilador no era perfecto, había sido adaptado del PLM85 (8085), pero Intel fue lo suficientemente realista para evitar el uso de un lenguaje totalmente dependiente del uso de la pila.

La solución adoptada fue sencillamente pasar los parámetros en áreas definidas de memoria. Así cada función o *procedure* tenía su propia área de memoria en la que recibía los parámetros, y devolvía los resultados. Si se utilizaba la RAM interna para el paso de parámetros, la sobrecarga de las llamadas a funciones era muy pequeña. Incluso utilizando RAM externa, siempre más lenta que la RAM interna, se consigue mayor velocidad que con una pila artificial.

El problema que tiene esta especie de "pila compilada" es que la re-entrancia no es posible. Esta aparentemente seria omisión, en la práctica no tiende a causar problemas con los típicos programas del 8051. Sin embargo las últimas versiones de C51 permiten la re-entrancia selectiva, es decir permiten que unas pocas funciones críticas sean re-entrantes, sin comprometer la eficiencia de todo el programa.

Otras consideraciones dignas de destacar para el C en un microcontrolador son:

1. Control de los periféricos internos y externos del chip.
2. Servicio de las interrupciones.
3. Hacer el mejor uso de los limitados conjuntos de instrucciones.
4. Soportar diferentes configuraciones de ROM/RAM.
5. Un alto nivel de optimización para conservar el espacio de código.
6. Control de la conmutación de registros.
7. Soporte para los derivados de la familia (87C751, 80C517 etc.).

El compilador Keil C51 contiene todas las extensiones para el uso del lenguaje C con microcontroladores. Este compilador C utiliza todas las técnicas apuntadas por Intel con su PLM51, pero añade características propias tales como la aritmética en coma flotante, la entrada/salida (I/O) con formato, etc. Se trata de la implementación del estándar ANSI C específico para los procesadores 8051.

Capítulo 2 - Fundamentos del compilador Keil C51 - La arquitectura del 8051.

2.1.- Introducción.

El compilador Keil C51 ha sido escrito para permitir que los programadores en lenguaje C obtengan código para el 8051, casi sin necesidad de aprender nuevos conceptos. Sin embargo, para sacar el mayor provecho de un µC, es conveniente saber algo sobre el hardware con el que se trabaja. Con el 8051, la decisión más importante está relacionada con el modelo de memoria a utilizar.

Para información general sobre el lenguaje C, y sobre la representación de números y cadenas, acudir a un libro estándar de C, tal como el K & R.

2.2.- Configuraciones de Memoria del 8051.

2.2.1.- Localización física de los espacios de memoria

Inicialmente la cosa más confusa sobre el 8051 es quizás la existencia de varios espacios de memoria, que comienzan en la misma dirección.

Otros µC, tales como el 68HC11, tienen una configuración de memoria mucho más sencilla, en la que solo existe un área de memoria de tipo Von Neuman, residente en uno o en varios chips.

Dentro del 8051 hay un espacio de RAM llamado DATA. Este espacio comienza en la dirección D:00 (el prefijo 'D:' indica segmento DATA) y termina en la dirección 0x7F (127 en decimal). Este área RAM puede utilizarse para almacenar las variables del programa. Se trata de una región direccionable directamente en la que pueden utilizarse instrucciones como 'MOV A,direcc'. Por encima de la dirección 0x7F se encuentran los registros de funciones especiales (SFR), que también son accesibles mediante direccionamiento directo. Sin embargo en algunos derivados del 8051 existe otro espacio de memoria entre las direcciones 0x80 y 0xFF, llamado IDATA (Indirect DATA), que sólo resulta accesible por direccionamiento indirecto (MOV A,@Ri). Para esta región que se solapa con los SFR se utiliza el prefijo 'I:'. El 8051 carece de estos 128 bytes del espacio IDATA, que se añadieron cuando apareció el 8052. Esta región resulta adecuada para la pila a la que siempre se accede indirectamente a través del apuntador de pila SP (Stack Pointer). Y para hacer las cosas más confusas, resulta que los 128 bytes de RAM comprendidos en las direcciones 0..0x7F también pueden ser accedidos indirectamente con la instrucción MOV A,@Ri

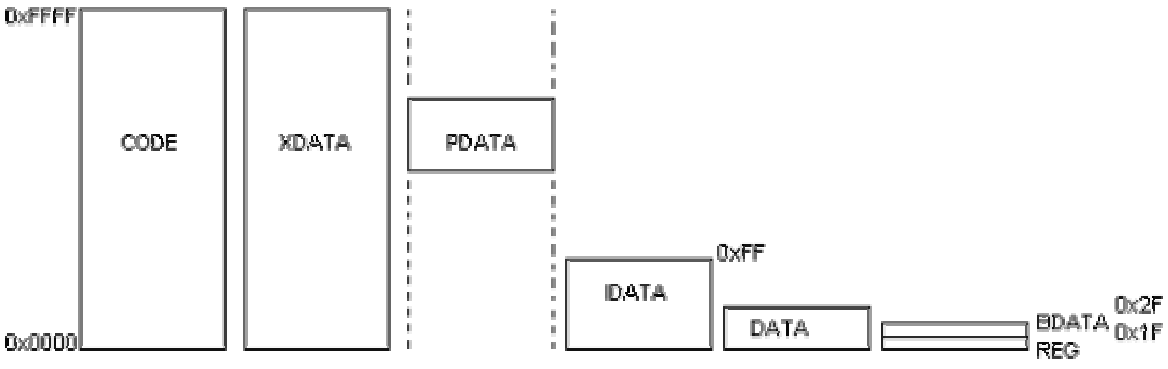

Fig.1. - Los espacios de memoria del 8051.

Un tercer espacio de memoria, el segmento CODE, también comienza en la dirección cero, pero está reservado para el programa o para almacenar valores constantes, ya que el µC no puede escribir en esta región. Se extiende desde C:0000 hasta C:0xFFFF (65536 bytes); parte de este área puede residir dentro del µC y la otra parte, si es necesaria, debe residir en chips externos de memoria EPROM o FLASH. El acceso al contenido del segmento CODE se realiza mediante el contador de programa PC (*Program Counter*) en los ciclos de búsqueda de instrucciones y con el DPTR (*Data Pointer*) para la lectura de los datos constantes.

La cuarta región de memoria, llamada XDATA reside en una RAM externa al µC. Comienza en X:0000 y se extiende hasta X:0xFFFF (65536 bytes). El acceso a esta región se realiza mediante el registro de 16 bits DPTR (*Data Pointer*). En esta región se puede seleccionar un pequeño espacio de 256 bytes, llamado PDATA (acceso paginado), al que se accede con un apuntador de 8 bits (registros R0 o R1). La dirección inicial de esta región PDATA viene dada por 16 bits, siendo los 8 bits de mayor peso los contenidos en el SFR P2, estando a 0 los 8 bits restantes.

Una pregunta inmediata es: "¿Cómo evita el 8051 que el acceso a un dato en D:00, realice también un acceso al dato X:0000?

La respuesta reside en el hardware del 8051: Cuando la CPU intenta acceder a D:00, habilita la RAM interna mediante una señal READ interna, que no provoca cambios en la patilla /RD, utilizada para leer la RAM externa.

```
MOV   A,40H     ; Lleva el valor de la dirección D:0x40 al acumulador.
```

Este modo de direccionamiento (**directo**) se utiliza en el modelo de memoria SMALL.

```
MOV   R0,#0A0H
MOV   A,@R0     ; Lleva el valor de la dirección I:0xA0 al acumulador
```

Este modo de direccionamiento se utiliza para acceder a la región de RAM interna de acceso **indirecto** situada por encima de la dirección 0x7F, aunque también es una forma alternativa de acceso a los 128 bytes situados debajo de la dirección 0x80.

Una variación de DATA es BDATA (bit data). Esta es un área de 16 bytes (128 bits) que se extiende desde D:0x20 hasta D:0x2F. Se trata de una región muy útil que permite el acceso normal a los bytes con las instrucciones MOV, y también permite el acceso a los bits mediante instrucciones especialmente orientadas al bit, como las siguientes:

```
SETB   20H.0  ; Pone a uno el bit 0 de la dirección D:0x20
CLR    20H.2  ; Pone a cero el bit 2 de la dirección D:0x20
```

El dispositivo externo EPROM o FLASH de memoria CODE no se habilita durante los accesos a la RAM. De echo, la memoria externa de tipo CODE sólo se habilita cuando una patilla del 8051 llamada PSEN (*Program Store Enable*) se pone a nivel bajo. El nombre CODE indica que la principal función de la EPROM o FLASH es almacenar el programa.

No existe colisión entre los accesos a la RAM XDATA y la EPROM-FLASH CODE, ya que el dispositivo externo XDATA sólo se activa a petición de dos patillas del 8051 llamadas /RD y /WR (*Read & Write*), mientras que el dispositivo externo CODE se activa exclusivamente cuando la patilla PSEN se pone a nivel bajo. Lógicamente el 8051 nunca activa a la vez las patillas PSEN y /RD o /WR.

Para acceder a la RAM XDATA existen instrucciones especiales (MOVX)

```
MOV   DPTR,#08000H
MOVX  A,@DPTR       ; "Lleva al acumulador el contenido de la posición de
                    ;  RAM externa cuya dirección está en DPTR (8000H)".
```

Este modo de direccionamiento se utiliza en el modelo de memoria LARGE.

```
MOV                                   R0,#080H                                  ;
MOVX  A,@R0     ;
```

Este es un modo de acceso alternativo a la RAM externa que se utiliza en el modelo de memoria COMPACT. La dirección realmente accedida es 0xYY80 siendo YY el contenido del SFR P2.

Un punto importante a recordar es que la patilla PSEN se activa durante la fase de búsqueda de instrucción, y con las instrucciones MOVC...(*Move Code*) usadas para la lectura de los datos constantes residentes en memoria de código. Por otro lado, las instrucciones MOVX... (*Move eXternal*) activan las patillas /RD o /WR, según que el movimiento sea hacia la CPU (lectura), o desde la CPU (escritura). La 'X' indica que la dirección no está dentro del 8051, sino que está contenida en un dispositivo eXterno que se habilita con las patillas /RD y /WR.

2.2.2.- Posibles modelos de memoria

Así como el programador de PC tiene que elegir entre los modelos de memoria *tiny, small, medium, compact, large y huge* para definir la segmentación de la RAM, el programador de 8051 tiene que decidir el modelo de memoria a utilizar, el cual determina el lugar de residencia del programa y de los datos.

C51 soporta actualmente las siguientes configuraciones de memoria:

ROM: el tamaño máximo del programa que permite generar C51 es de 64K, sin embargo es posible aumentarlo hasta 1MB mediante el modelo BANKED descrito más abajo. Todos los elementos que se deseen almacenar en la EPROM/ROM, tales como constantes, tablas etc., deben declararse como *code*.

RAM: Se admiten tres modelos de memoria, SMALL, COMPACT y LARGE

1. **SMALL**: todas las variables y segmentos de paso de parámetros se guardan en RAM interna.
2. **COMPACT:** las variables se almacenan en una región de RAM externa de acceso paginado y de tamaño máximo 256 bytes. El acceso a las mismas se hace indirectamente por medio de instrucciones "MOVX A,@R0". Los registros internos del 8051 se usan para variables locales y parámetros de función.
3. **LARGE:** las variables se almacenan en memoria externa, y se accede a ellas con instrucciones "MOVX A,@DPTR", lo que permite disponer de un espacio máximo de 64Kbytes para las variables. Los registros internos del 8051 se usan para variables locales y parámetros de función.

Modelo BANKED: Con este modelo el código puede ocupar hasta 1MB utilizando algunas patillas de un puerto del 8051, o un *latch* mapeado en memoria. Estos hilos se utilizan para paginar la memoria por encima de la dirección 0xFFFF. Dentro de cada bloque de 64KB debe existir un bloque común (COMMON) para hacer posibles las llamadas a funciones que se encuentren en bancos distintos.

Además de la elección del modelo de RAM, es posible utilizar un modelo globalmente y forzar que ciertas variables y objetos residan en otros espacios de memoria.

Se tratará esta técnica más tarde.

2.2.3.- Elección del mejor modelo de memoria

Al disponer de tres modelos de memoria para la RAM, se necesitan criterios para seleccionar el modelo óptimo. En aplicaciones monochip sólo se puede utilizar el modelo SMALL. Si se dispone de memoria RAM externa, es posible elegir entre los modelos SMALL (MOV A,Direct), COMPACT (MOVX A,@R0) y LARGE (MOVX A,@DPTR).

Aunque es posible cambiar de modelo de memoria a lo largo de un proyecto, no es una práctica recomendable.

SMALL: RAM total 128 bytes (8051/31)

En el modelo SMALL, las variables se almacenan en la RAM interna del 8051, por lo que el acceso a las mismas es muy eficiente. Debido al limitado tamaño de la RAM del 8051 (128 bytes), y a que en ella residen los registros R0..R7 del procesador y la pila del programa, el espacio disponible para variables es muy reducido. Pese a ello, el modelo SMALL es válido para la mayor parte de las aplicaciones, debido a que el *linker* maneja la escasa RAM interna del 8051 con técnicas de solapamiento (OVERLAY). De esta manera, el área destinada a los parámetros y a las variables locales de distintas funciones se solapan, si éstas no se llaman mutuamente ni de forma recursiva.

Véase el Capítulo 8 "Solapamiento de la memoria data realizado por el L51"para más información sobre la técnica de solapamiento.

El modelo SMALL es el más adecuado para aplicaciones en las que el tiempo de ejecución es un factor crítico, ya que el 8051 accede a su RAM interna con mayor rapidez que a cualquier otra área de memoria. Para aprovechar las características ventajosas de este modelo, incluso en programas muy grandes, el programador debe procurar que los arrays y objetos de tamaño grande, así como las variables a las que se accede con poca frecuencia, residan en RAM externa. Por el contrario las variables de acceso frecuente, y de pequeño tamaño deben residir en RAM interna.

COMPACT: RAM total 256 bytes fuera del chip, 128 o 256 bytes dentro del chip.

Puede ser el mejor modelo en aplicaciones en las que el sistema operativo hace uso de la RAM interna, o en las que la pila adquiere un tamaño elevado. Las variables se almacenan en RAM externa pudiendo tener un tamaño máximo de 128 bytes. El acceso a las mismas se hace mediante instrucciones MOVX A,@R0, lo que proporciona una velocidad de acceso intermedia entre el modelo SMALL (más rápido) y el modelo LARGE (más lento).

LARGE: RAM total 64KB fuera del chip, 128 o 256 bytes dentro del chip.

Las variables se almacenan en RAM externa pudiendo tener un tamaño máximo de 64 Kbytes. Se accede a las mismas mediante instrucciones MOVX A,@DPTR, lo que proporciona el acceso más lento de todos los modelos de memoria.

Resumiendo: el 8051 dispone de seis espacios de memoria para almacenamiento de datos. Al definir una variable se puede especificar el espacio de memoria elegido para la misma, por medio de las extensiones al lenguaje C para el 8051: data, bdata, idata, pdata, xdata y code. Si al definir una variable se omite el espacio de memoria asignado a la misma, el compilador C51 utiliza automáticamente el tipo de memoria correspondiente al modelo (SMALL, COMPACT o LARGE) utilizado.

Cada tipo de memoria tiene sus pros y sus contras. Aquí se ofrecen algunas recomendaciones para hacer el mejor uso de los mismos.

DATA: 128 bytes; área utilizada por el modelo SMALL

- ***El mejor para:*** Datos a los que se accede con frecuencia, variables usadas por rutinas de interrupción, variables de rutinas re-entrantes.
- ***El peor para:*** Arrays y estructuras de tamaño de medio a grande.

IDATA; No dependiente del modelo utilizado

- ***El mejor para:*** Acceso rápido a arrays y estructuras de tamaño medio (unos 32 bytes cada uno, sin sobrepasar los 64 bytes). Ya que el acceso a este tipo de datos suele realizarse indirectamente, mediante punteros, éste tipo de memoria es el mejor para los mismos. También es un buen lugar para la pila, a la que siempre se accede indirectamente.
- ***El peor para:*** Grandes arrays de datos.

CODE: 64K bytes

- ***El mejor para:*** Constantes y grandes tablas, además de para el código del programa, ¡No faltaba más!
- ***El peor para:*** ¡Variables!

PDATA: 256 bytes; área utilizada por el modelo COMPACT

- ***El mejor para:*** Accesos a datos que requieran una velocidad intermedia, así como a matrices y estructuras de tamaño moderado.
- ***El peor para:*** Arrays y estructuras cuyo tamaño supere los 256 bytes. Datos a los que se accede con mucha frecuencia, etc..

XDATA; área utilizada por el modelo LARGE

- ***El mejor para:*** Arrays y estructuras cuyo tamaño supere los 256 bytes. Variables a las que se accede con poca frecuencia.

- ***El peor para:*** Datos a los que se accede con mucha frecuencia, etc..

2.2.4.- Selección del modelo de memoria - Uso de #Pragma

La selección del tipo de memoria global se realiza incluyendo la línea #pragma SMALL (o COMPACT o LARGE) como primera línea de un fichero C. El compilador C51 utiliza por omisión de la directiva #pragma, el modelo SMALL. Este modelo puede utilizarse en casi el 100% de las aplicaciones, si se tiene la precaución de forzar las variables de tamaño grande, y las variables a las que se accede rara vez, en las áreas PDATA y XDATA.

Véase el capítulo 2 para detalles sobre ubicación de variables.

Nota sobre el uso del modelo COMPACT

El modelo COMPACT hace ciertas suposiciones sobre el estado del puerto P2. El espacio XDATA se direcciona mediante instrucciones MOVX que ponen los 16 bits del registro DPTR en los puertos P2 y P0. El modelo COMPACT utiliza el registro R0 como un apuntador de 8 bits, cuyo contenido se pone en el puerto P0 cuando se ejecuta la instrucción MOVX A,@R0. El puerto P2 queda bajo control del usuario para el acceso paginado a la RAM externa. El compilador no tiene información sobre P2, y a menos que se le asigne explícitamente un valor, su contenido será indefinido, aunque generalmente será 0xFF. El linker tiene la tarea de combinar las variables XDATA y PDATA, y si no se le informa adecuadamente, coloca el área PDATA en dirección 0. Por lo tanto el programa COMPACT no funcionará.

Es por tanto esencial asignar a PPAGE en el fichero "startup.a51" el valor adecuado para P2, y poner PPAGEENABLE a 1 para habilitar el modo paginado. La asignación del valor de PPAGE también puede hacerse por medio del control PDATA(ADDR) al realizar la llamada al linker como en:

L51 module1.obj, module2.obj to exec.abs PDATA(0)XDATA(100H)

Notar que el área normal XDATA comienza ahora en 0x100, por encima de la página cero usada para PDATA

2.3.- ESPECIFICACIÓN LOCAL DEL MODELO DE MEMORIA.

2.3.1.- Un aspecto a vigilar en proyectos multi-modelo

C51 permite asignar modelos de memoria a funciones individuales. Dentro de un mismo módulo, las funciones pueden declararse como SMALL, COMPACT o LARGE así:

```
#pragma COMPACT
/* Una función con modelo SMALL */
void fsmall(void) small {
   printf("HELLO") ;
   }
/* Una función con modelo LARGE */
void flarge(void) large {
   printf("HELLO") ;
   }
main() {
   fsmall() ;  // Llamada a función small.
   flarge() ;  // Llamada a función large.
   }
```

2.3.1.- Un aspecto a vigilar en proyectos multi-modelo

Supongamos que un programa C51 utiliza el modelo COMPACT para todas sus funciones excepto para las funciones de interrupción, para las que se desea utilizar el modelo SMALL que resulta más rápido. En estos casos, de no actuar correctamente, el linker puede emitir mensajes del tipo MULTIPLE PUBLIC DEFINITION refiriéndose por ejemplo a la función putchar().

Ello se debe a que en los módulos compilados como COMPACT, C51 crea referencias a funciones de la librería COMPACT, mientras que los módulos SMALL acceden a las funciones de la librería SMALL. De esta forma el linker L51 puede encontarse con dos putchars() de distintas librerías.

La solución en este caso consiste en utilizar globalmente el modelo de memoria COMPACT y seleccionar localmente un modelo diferente para ciertas funciones.

Ejemplo:

```
#pragma COMPACT
    void fast_func(void) small{
    /* código */
    }
```

Capítulo 3 - Declaración de constantes y variables

3.1.- Constantes

Un requisito básico para la escritura de cualquier programa, es conocer la ubicación de los datos del mismo. Las constantes son las más sencillas de ubicar, deben residir en el área de código (EPROM), o como constantes en RAM, inicializadas en tiempo de ejecución, cuyo valor lógicamente no debe cambiar mientras dure el programa.

El último caso es la situación habitual con los programas para PC generados con Microsoft C o Borland C, sin embargo en las aplicaciones con el 8051, lo más adecuado es colocar las constantes en ROM, con lo cual se ahorra espacio en RAM. Ejemplos de constantes en EPROM son:

```
unsigned char code temperat = 0x02 ;
unsigned char code tabla[5] = {'1','2','3','4''} ;
unsigned int code  presión  = 4 ;
```

Notar que *const* no significa *code*. Los objectos declarados *const* residirán en el área de memoria de datos correspondiente al modelo de memoria utilizado.

Para colocar una tabla grande en el área CODE, la declaración debe ser:

```
unsigned char code TABLA_2[] = {
            0x00,0x00,0x00,0x09,0x41,0x80,0xC0,0xFF,
            0x00,0x00,0x13,0x1A,0x26,0x33,0x80,0xFF,
            0x00,0x00,0x00,0x09,0x41,0x80,0x66,0x66,
            0x00,0x00,0x00,0x09,0x41,0x80,0x66,0x66,
            0x00,0x00,0x00,0x00,0x4D,0x63,0x66,0x66,
            0x00,0x00,0x00,0x02,0x4D,0x63,0x66,0x66,
            0x00,0x00,0x00,0x05,0x4A,0x46,0x40,0x40,
            0x00,0x00,0x00,0x08,0x43,0x43,0x3D,0x3A,
            0x00,0x00,0x00,0x00,0x2D,0x4D,0x56,0x4D,
            0x00,0x00,0x00,0x00,0x21,0x56,0x6C,0x6F
        } ;
```

Con objetos grandes como el de arriba es preciso controlar el espacio de memoria. En particular cuando se trabaja con el modelo SMALL es muy fácil llenar toda la RAM interna con una sola tabla.

La definición de constantes en RAM puede realizarse así:

```
unsigned char factor_de_escala = 128    ;
unsigned int constante_de_fuel = 0xFD34 ;
```

En este caso es mejor hablar de variables inicializadas - véase el capítulo 3 "Inicialización de variables" en lugar de constantes, ya que su valor puede modificarse a lo largo del programa

3.2.- Variables

Naturalmente todas las variables residen en RAM, la configuración de la RAM puede verse en el Capítulo 2 "Localización física de los espacios de memoria".

3.2.1.- Variables no inicializadas

La directiva '#pragma SMALL' determina el modelo de memoria global. En este caso, todas las variables se colocan en RAM interna. Sin embargo, se puede forzar la residencia de algunas variables en áreas específicas de memoria, como sigue:

```
#pragma SMALL
   .
   .
   unsigned char xdata velocidad_motor ;
   char xdata gran_array_de_variables[192] ;
```

Ahora la variable velocidad_motor se coloca en RAM externa. En el caso del array, no es posible colocarlo en RAM interna, debido a que el tamaño máximo de la misma es de 128 bytes.

Otro ejemplo es:

```
   .
#pragma LARGE
   .
   .
   .
   función(unsigned char data para1)
   {
   unsigned char data variable_local ;
   .
   .
   .
   }
```

Aquí, los parámetros de la función se colocan en RAM interna para reducir el tiempo de llamada a la función. En caso de omitir la palabra 'data', los parámetros de la función residirían en XDATA, tal como corresponde al modelo de memoria LARGE.

En este caso puede ser mejor declarar la función como SMALL, aunque en el resto del programa se utilice el modelo LARGE. Esta técnica se utiliza para producir unas pocas funciones muy rápidas en programas grandes obtenidos con el modelo LARGE.

Si el sistema dispone de RAM externa paginada en el puerto P0, la directiva apropiada es *pdata*.

Véanse las notas del capítulo2 "Elección del mejor modelo de memoria" para ubicación óptima de variables.

3.2.2.- Inicialización de Variables

El lenguaje C garantiza que en ausencia de una inicialización explícita, sólo las variables externas y las estáticas toman valor inicial cero. Si se desea un valor distinto a cero, o se quiere

forzar un valor inicial para una variable que no sea externa, ni estática, es muy útil declarar e inicializar las variables al mismo tiempo, tal como se muestra a continuación:

```
unsigned int velocidad_motor = 0x20 ;

 función()
   {
   .
   .
   }
```

Aquí se escribe el valor 0x20 en la variable antes de que cualquier función tenga acceso a la misma. Para lograrlo, el compilador reúne todos los valores iniciales de las variables en una tabla, y pasa el control del programa al módulo "startup.obj", que copia los valores de la tabla en las direcciones apropiadas de RAM, llamando seguidamente a la función main().

Es responsabilidad del usuario modificar el fichero en ensamblador "startup.a51" para informar al compilador del tamaño y dirección inicial de la memoria RAM disponible en su sistema hardware. Si se utiliza el modelo de memoria LARGE los parámetros a cambiar en "startup.a51" son: XDATASTART y XDATALEN.

3.3.- WATCHDOG E INICIALIZACIÓN DE VARIABLES

En programas que inicialicen muchos datos, puede suceder que el proceso de inicialización de los mismos dure más que el tiempo de refresco del *watchdog*. En este caso, la CPU puede resultar reseteada antes de alcanzar la función main(). Para evitarlo, hay que modificar el fichero en ensamblador INIT.A51, en el directorio \C51\LIB.

Este fichero contiene una macro especial llamada WATCHDOG, que el usuario debe rellenar con el código de refresco del watchdog. La macro se inserta automáticamente en cada bucle de inicialización del fichero INIT.A51

3.4.- VARIABLES EN C51.

3.4.1.- Tipos de datos.

Los tipos de datos admitidos por C51 son:

```
TIPO            TAMAÑO          RANGO DE VALORES
bit           = 1 bit           0 ó 1
char          = 8 bits          -128 a +127
unsigned char = 8 bits          0 a 255
int           = 16 bits         -32768 a +32767
unsigned int  = 16 bits         0 a 65535
long          = 32 bits         -2147483648 a +2147483647
unsigned long = 32 bits         0 a 4294967295
float         = 32 bits         +/-1.175494E-38 a +/-3.402823E+38
puntero       = 24/16/8 bits    Dirección de una variable

Son declaraciones típicas las siguientes:
    unsigned char xdata tensión_batería ;
    int idata factor_de_corrección ;
    bit flag_1 ;
```

(Nota: las variables bit siempre residen en el área de memoria 'bdata' direccionable a nivel de bit del 8051 - véase capitulo 2 "Localización física de los espacios de memoria")

Para un procesador como el 8086, el tipo int es el más común, ya que al tratarse de un procesador de 16 bits, obtiene el máximo de eficiencia cuando trabaja con valores de 16 bits. Sin embargo, con el 8051, el tipo de dato a utilizar debe ser lógicamente el char. Por otro lado, ya que el 8051 no dispone de instrucciones para la aritmética con signo, es conveniente trabajar, siempre que ello sea posible, con tipos *unsigned* en lugar de con tipos *signed*. Si no se siguen estas reglas, los programas resultantes tendrán mayor tamaño y funcionarán con mayor lentitud.

El µC Siemens 80C537 dispone de un co-procesador matemático integrado, que le permite realizar instrucciones con números enteros y operaciones complejas como la división de un número de 32 bits, entre un número de 16 bits. El compilador Keil C51 dispone de una librería específica para el 80C537 que saca partido de las características especiales de este procesador.

3.4.2.- Bits de funciones especiales.

El programador en lenguaje ensamblador del 8051, encuentra que el lenguaje ANSI C no le ofrece facilidades para operar con los bits del área BDATA, y le obliga a trabajar con máscaras para comprobar el estado de los bits. Sin embargo, el compilador C51 permite ubicar variables en el área de RAM interna de acceso a nivel de bit y a nivel de byte, y manejar directamente desde C las instrucciones de manipulación de bits del 8051.

Veamos como ejemplo la comprobación del bit de signo de un char.

En primer lugar, se fuerza la residencia del char en el área "bdata":

```
char bdata test ;
```

el bit de signo se define como sigue:

```
sbit signo = test ^ 7  ;
```

Y se usa así:

```
void main(void) {
     test = -1 ;
     if(test & 0x80) { //Comprobación convencional con
máscara&
        test = 1 ;      //tratamiento si número negativo
     }
     if(signo == 1)  { //Uso del bit de signo
        test = 1 ;      //tratamiento si número negativo
        }
      }
```

Resultado en ensamblador:

```
RSEG  ?BA?T2
           test:   DS    1
signo EQU (test+0).7

; void main(void) {
```

```
main:
;       test = -1 ;
MOV  test,#0FFH
;       if(test & 0x80) {//Comprobación convencional con
máscara&
MOV  A,test
JNB  ACC.7,?C0001
;          test = 1 ;      // tratamiento si número negativo
MOV  test,#01H
;       }
?C0001:
;       if(signo == 1)  { // Uso del bit de signo
JNB  signo,?C0003
;          test = 1 ;      // tratamiento si número negativo
MOV  test,#01H
;          }
;        }
?C0003:
RET
```

La definición del bit de signo mediante la extensión *sbit*, hace que el compilador utilice la instrucción JNB, lo cual proporciona mayor rapidez que el uso de máscaras &.

En en supuesto que se desee comprobar el signo de un valor int, se debe tener en cuenta que el el 8051 almacena el byte alto en la dirección de memoria más baja. Por ello, el bit 7 es el bit de mayor peso del byte alto, mientras que el bit 15 es el bit de mayor peso del byte bajo de la variable int.

3.4.3.- Conversión entre tipos.

Una de las principales fuentes de error en C, consiste en ignorar las implicaciones de los tipos de datos en los cálculos y comparaciones. En el siguiente ejemplo:

```
unsigned char x, y, z ;

 x = 10 ;
 y = 5  ;

 z = x * y ;
```

El resultado es z = 50

Sin embargo:

```
x = 10 ;
y = 50 ;

z = x * y ;
```

hace que z = 244. La solución verdadera 500 (0x1F4) no se logra, debido a que z no puede almacenarla. La solución consiste en hacer que z sea un unsigned int. Sin embargo, siempre es una buena idea hacer explícito el forzado de tipo (casting) de los operandos unsigned char a int así:

```
unsigned char x, y, z ;
```

```
z = (unsigned int) x * (unsigned int) y ;
```

Aunque C51 promueve automáticamente los char a int, lo mejor es no confiar en ello, ya que en un pequeño microcontrolador siempre hay que cuidar el tamaño exacto de los datos.

3.4.4.- Una solución no-ANSI para la comprobación de valores.

Una situación muy corriente se produce cuando se suman dos bytes y se limita el resultado máximo de la suma al valor 255. Ya se ha comentado que con el 8051, si se desea conseguir la máxima velocidad conviene evitar el uso de variables int. Por otro lado, si la suma de los dos bytes supera el valor 255, es necesario utilizar enteros.

Seguidamente se presentan dos ejemplos. En el primero se utiliza un enfoque ANSI, y puede verse el buen trabajo realizado por el compilador C51. En el segundo ejemplo, mucho más rápido se emplea una solución no ANSI:

```
; #include <reg51.h>
; unsigned char x, y, z ;
;
; /***Sumar dos bytes y comprobar resultado supera el valor
255***/
;
; void main(void) {

RSEG  ?PR?main?T2
USING 0
main:
;     if(((unsigned int)x + (unsigned int)y) > 0xff)
MOV  A,x
ADD  A,y
MOV  R7,A
CLR  A
RLC  A
MOV  R6,A
SETB C
MOV  A,R7
SUBB A,#0FFH
MOV  A,R6
SUBB A,#00H
JC   ?C0002
;         z = 0xff ;   // versión ANSI C
MOV  z,#0FFH
;     }
?C0002:
RET
```

En este segundo ejemplo, se comprueba el valor del flag de acarreo CY, eliminando la necesidad de realizar operaciones con enteros, ya que si el resultado supera el valor 255, el bit CY toma valor 1. Por supuesto, se trata de una solución no conforme al estándar ANSI C.

```
; #include <reg51.h>
;
; unsigned char x, y, z ;
;
; /*** Sumar dos bytes y comprobar si el resultado supera el valor
255 ***/
```

```
; void main(void) {
main:
;     z = x + y;
MOV  A,x
ADD  A,y
MOV  z,A
;     if (CY == 1)
JNB  CY,?C0002
;        z = 0xff ;   // Solución con el flag CY del 8051
MOV  z,#0FFH
;     }
?C0002:
RET
```

Si se suman dos enteros y se desea comparar si el resultado supera el valor 65535 (0xffff), la situación se complica ya que hay que utilizar valores long, que ocupan 32 bits. Aún en este caso, se puede comprobar el valor del bit CY después de realizar la suma de los bytes altos de los dos enteros. En sistemas de altas prestaciones con el 8051, la pérdida de portabilidad puede tolerarse. Desafortunadamente la portabilidad de los programas siempre compromete las prestaciones de los mismos.

Capítulo 4 - Disposición y estructura de los programas

4.1.- PROGRAMACIÓN MODULAR EN C51

Salvo en los programas triviales, el trabajo desarrollado por el software se compone de pequeñas tareas, que deben identificarse antes de empezar a escribir código. Al igual que un sistema electrónico está formado por varios módulos que realizan funciones diferenciadas, un sistema software se construye mediante tareas discretas. En el ejemplo electrónico, cada módulo debe ser diseñado y perfeccionado individualmente antes de construir el equipo. Con el software, las tareas son bloques individuales que se deben reunir para lograr el objetivo final.

De esta forma, el programa completo tiene una estructura modular que le proporciona un formato racional. Las tareas son los bloques más grandes que pueden identificarse en el programa. Estas se construyen con módulos, o ficheros de código fuente creados con un editor. A su vez los módulos, en el caso del lenguaje C, se construyen utilizando funciones.

La base de la programación modular reside en agrupar las secciones de software según la función a que se encuentran asociadas.

Así un sistema para el control de un motor estaría formado por las siguientes tareas:

```
Tarea 1
Proporciona la temporización para los chispazos de los inyectores

Tarea 2
Proporciona pulsos de anchura controlada para la inyección del fuel

Tarea 3
Permite ajustar los parámetros desde un terminal
```

A su vez la Tarea 1, estaría formada por los siguientes módulos:

```
Tarea 1, Módulo 1
Determinación de la posición y velocidad del árbol de levas

Tarea 1, Módulo 2
Obtención del ángulo de disparo desde una tabla de valores

Tarea 1, Módulo 3
Medida de la carga del motor
```

El módulo 3 contiene una función C que utiliza un convertidor A/D para leer la tensión proporcionada por un sensor, siguiendo una secuencia fija. En el módulo 1, una función de interrupción asociada a una patilla de captura del µC, calcula la velocidad del motor y genera un pulso para la bobina de encendido. En el módulo 2 se encuentra el bucle principal que calcula el ángulo de disparo, basándose en la información proporcionada por otros módulos, sobre la velocidad y la carga del motor. Lógicamente las funciones que recolectan datos, deben pasarlos a las funciones que los procesan y a los que generan las señales. Ello exige que funciones residentes en diferentes módulos o ficheros, tengan la posibilidad de intercambiar valores. En este caso, el flujo de datos sería:

```
Módulo 1                        Módulo 2                    Módulo 3

Sensor
   |
Entrada de captura                                      Sensor de presión
   |                                                          |
Velocidad de motor ----->  Hallar ángulo de avance  <--Carga del motor
   |                              |
Angulo de avance   <----------------|
   |
Salida de comparación
   |
Bobina de encendido
```

Generalmente, las variables se declaran en el módulo que les asigna un valor por primera vez. Por ello la variable carga_motor estaría definida en el módulo 3 que es quien le proporciona los datos de entrada. La declaración de los datos sería:

```
#define UCHAR unsigned char
```

```
Módulo_1.c                    Módulo_2.c                    Módulo_3.c

/*Declaraciones globales*/ /*Declaraciones globales*/ /*Declaraciones
globales */

UCHAR velocidad_motor       UCHAR avance               UCHAR
carga_motor

/* Referencias externas */ /* Referencias externas */

extern UCHAR avance        extern UCHAR velocidad_motor, carga_motor
```

Es importante señalar que cuando un módulo hace referencia a una variable definida en otro módulo, debe declarar el tipo de esa variable utilizando el prefijo *extern*.

Ahora, con un programa que se extiende por varios módulos, surge el problema de la comunicación y la llamada a funciones entre diferentes módulos.

La siguiente sección ilustra cómo se realiza el enlazado (linkado) entre módulos.

4.2.- ACCESO A VARIABLES EN PROGRAMAS MODULARES

Una aplicación típica, escrita en C51 puede residir en 5 módulos o ficheros fuentes. Cada fichero contendrá un número de funciones (subrutinas) que operan y utilizan variables en RAM. Cada función individualmente recibirá sus datos de entrada y devolverá sus resultados según el mecanismo de paso de parámetros establecido en C51. Además cada función utiliza variables temporales para almacenar los resultados intermedios. Cuando se programa en ensamblador, se acostumbra a reservar un lugar en memoria a cada variable (incluso las temporales), y hacerlas accesibles a todas las subrutinas.

Este método es muy ineficiente y limitaría seriamente la potencia de los programas C, debido a que la RAM interna, resultaría insuficiente para muchos programas, y además se perdería el claro mecanismo de entradas y salidas de funciones, propio de los lenguajes de alto nivel. Por otro lado, la totalidad del programa debería escribirse en un único fichero fuente, al igual que se hacía en ensamblador. El tamaño del programa resultante alargaría el proceso de compilación, y

se perdería el ideal de dividir los programas en pequeños trozos fáciles de entender. Así los programas se convertirían en bloques monolíticos, cuyos listados llenarían muchísimas hojas de papel.

Por lo tanto, dentro de un programa debe existir una organización jerárquica de variables y funciones, los bloques funcionales deben identificarse y reunirse en módulos individuales, y para que ello sea posible, debe explotarse la habilidad de acceso a variables externas y a funciones de otros módulos. Lo que sigue puede ayudar a comprender:

```
MODULO1.c: ********************************
   UCHAR global1 ;     (1)
   UCHAR global2 ;
   extern UCHAR ext_función(UCHAR) ;     (2)

   UCHAR int_función(UCHAR x)       (3)
   {
   unsigned int temp1  ;     (4)
   UCHAR temp2 ;
   temp1  = x * x ;
   temp2  = x + x ;

   x = temp1/temp2 ;

   return(x)     (5)
   }

/* Programa principal */
   void main(void)     (6)
   {
   UCHAR local1 ;     (4)
   UCHAR local2 ;
   local2 = int_función(local1) ;     (7)
   local1 = ext_función(local2) ;     (8)

   }
final de MODULO1.c ************************
```

```
MODULO2.c: ********************************
   extern UCHAR global1 ;     (9)

   UCHAR ext_función(UCHAR y)
   {
   UCHAR temp = 2;
   static UCHAR special ;     (10)

   special++ ;
   y = temp * global1 ;

   return(y) ;
   )
```

Línea (1): declara variables accesibles desde cualquier lugar del programa. En el caso ideal no deberían utilizarse estas variables **externas o globales**, pero son imprescindibles cuando una interrupción debe actualizar un valor que necesita el programa principal.

Línea (2): declara una función externa, o no definida en ese módulo. Con ello se permiten las llamadas posteriores a la función externa.

Línea (3): define una función que será llamada desde otras funciones de este módulo. La variable UCHAR x, sólo existe mientras el programa esté ejecutando esta función, y desaparecerá cuando se salga de ella. Es conveniente definir las funciones antes de que otras funciones las llamen (como hace main en este caso).

Línea (4): como sucedía antes con la variable x, estas variables **locales** se utilizan para guardar resultados intermedios de la función, y sólo existen mientras se ejecuta la función. Al salir de la función, el espacio que ocupaban estas variables se puede asignar a otras variables. Sin embargo las variables locales definidas en main() siempre existen, ya que todo el programa C está contenido en main().

Línea (5): permite que la función devuelva un valor a la función que la llamó. Una vez que se regrese a main(),este valor se asigna a la variable "local2".

Línea (6): define el comienzo del un programa C. Sin embargo, antes de que el programa llegue a main(), se ejecuta la rutina contenida en "STARTUP.A51", que comienza en C:0000, o posición del vector reset. La función main() no recibe ningún parámetro.

Línea (7): llama a la función descrita arriba, y le pasa el valor "local1".

Línea (8): como en 7, solo que la función llamada reside fuera del módulo.

Línea (9): está vinculada a la Línea(1). Hace que la variable "global1" sea visible en MODULO2.C.

Línea (10): declara una variable que es local a esta función, pero que no debe ser destruida al salir. Así se comporta como una variable global, salvo que ninguna otra función puede utilizarla. Si se hubiera definido encima de la función, su accesibilidad se extendería a todas las funciones del MODULO2 que le sigan, haciéndola invisible a otros módulos.

El enlazado de los nombre de variables y funciones entre los diferentes módulos, lo realiza el linker L51. Este aspecto se trata en la Capítulo 8.

4.3.- LA CONSTRUCCIÓN DE UN PROGRAMA MODULAR

Ya se ha explicado anteriormente la necesidad de construir programas modulares. Aquí se tratará sobre los aspectos prácticos de la construcción de software documentado y fácilmente mantenible, junto a un truco para desarrollar programas C utilizando compiladores como el Keil C51.

4.3.1.- El problema

Uno de los más sencillos programas C consitiría en:

```
/* Módulo de inicialización del puerto serie */ /* V24IN537.C */
void v24ini_537(void)
   {
   /* Código inicialización del puerto serie */
```

```
   }

/* Módulo con el programa principal */ /* MAIN.C */
/*  Definiciones Externas */

extern void v24ini_537(void) ;

void main(void) {
   v24ini_537() ;
   while(1) {
      printf("Tiempo = ") ;
      }
```

Este pequeño programa tiene un solo propósito, imprimir un mensaje todavía incompleto a un terminal conectado al puerto serie. Obviamente, un solo módulo, o fichero fuente es suficiente para almacenar el programa completo.

Cualquier programa real contiene mayor funcionalidad que el anterior. La reacción natural consiste en añadirle funciones hasta lograr la funcionalidad deseada. Pero si no se toman acciones en contra, se puede conseguir un fichero enorme con docenas de funciones e interrupciones y quizás con cientos de variables públicas.

Se tardaría mucho tiempo en compilar el programa, y la menor modificación del mismo que normalmente obliga a re-compilar, haría aún mas lento el proceso. Un programa monolítico indica que no ha existido una planificación suficiente, además es difícil de mantener y su código a primera vista no ofrece muchas garantías.

El paso siguiente en el desarrollo del programa anterior consiste en añadir algún medio para generar la temporización:

```
/* Módulo con la inicialización del Timer0 */ /* T0INI537.C */

   void timer0_init_537(void) {
        /* Enable Timer 0 Ext0 interrupts */
        } /*init_timer_0*/

/* Módulo con la rutina de atención al Timer0 */ /* RLT_INT.C */
/* Declaración de variables locales */
/* Estructura para el reloj */

struct time { UCHAR msec ;
              UCHAR sec  ; } ;

/* Creación de la estructura en XDATA */

struct time xdata clock ;

bit clock_reset_fl = 0   // Flag para indicar a la interrupción
                         // del timer 0, que ponga el reloj a cero

/* Referencias externas */

extern bit clock_run_fl; // Flag para indicar a la interrupción
                         // del timer 0 que detenga el reloj

/***  ATENCION A LA INTERRUPCION DEL TIMER 0  ***/
```

```
   void timer0_int(void) interrupt 1 using 1 {
     if(clock.msec++ == 1000) {
        clock.sec++ ;
        if(clock.sec == 60) {
           clock.sec = 0 ;
           }
        }
     }
```

Para que este módulo sea útil, hay que cambiar el bucle principal a:

```
/* Módulo con el programa principal */ /* MAIN.C */

#include <reg517.h>

/* Definiciones externas */

extern void v24ini_537(void) ;
extern void timer0_init_537(void) ;

/* Estructura para el reloj */

struct time { UCHAR hours ;
              UCHAR mins ;
              UCHAR secs ;
              UCHAR msec  ; } ;

/* Referencia a estructura XDATA en otro módulo */

extern struct time xdata clock ;
extern bit clock_reset_fl // Flag para indicar a la interrupción
                          // del timer 0, que ponga el reloj a cero
/* Declaración de variables locales */
bit clock_run_fl  ;     // Flag para indicar a la interrupción
                        // del timer 0 que detenga el reloj
void main(void) {
   v24ini_537() ;
   timer0_init_537() ;
   while(1) {
      printf("Tiempo = %d:%d:%d:%d",clock.hours,
                                    clock.mins,
                                    clock.secs,
                                    clock.msecs) ;
      }
   if((P1 & 0x01) != 0) {
      clock_run_fl = 1 ; // Si se ha pulsado start clock
      }
   else {
      clock_run_fl = 0 ; // Si se ha soltado stop clock
      }
   if((P1 & 0x02) != 0) {
      clock_reset_fl = 1 ; // Si se ha pulsado clear clock
      }
   }
```

4.3.2.- Matenimiento de los vínculos inter-módulo

El programa anterior se ha construido de una forma modular, con cada bloque funcional dentro de un módulo separado. Sin embargo, aún en este pequeño programa se hace evidente el problema del mantenimiento. Está claro que, cada vez que se añade una nueva variable o función, es necesario editar dos ficheros, el módulo que contiene la definición y los módulos que se refieren a ella. Si además se utilizan nombres largos, aparecen errores tipográficos que originan grandes pérdidas de tiempo.

En programas largos, con muchas funciones y variables externas, el área global que precede al código ejecutable puede quedar en desorden y muy abultada. Se puede argumentar que añadir referencias externas al principio de un módulo es una buena práctica, porque se tiene un control preciso de las variables utilizadas. No obstante, es preferible la práctica utilizada con frecuencia, que consiste en incluir en cada módulo fuente, un fichero que contenga las referencias externas a cada variable global o función, independientemente de que el módulo en cuestión tenga necesidad de ellas.

Este método conduce a una situación indeseable, en la que un módulo fuente define una función y encuentra una referencia externa a la misma en el fichero general incluido a comienzo del mismo.

Una solución puede consistir en disponer de ficheros *include* específicos. Así por cada módulo ".c", se crearía un segundo fichero ".h". Este fichero auxiliar debería contener los prototipos de las funciones del módulo ".c", las declaraciones de variables globales, y las referencias externas a las mismas funciones y variables globales. Se trata de un concepto similar al de los ficheros ".h" y las librerías estándar de cada compilador C. El truco, consiste en utilizar compilación condicional para evitar que las declaraciones de los elementos, y las mismas declaraciones externas resulten visibles a la vez.

Así al incluir el fichero ".h" en el fichero ".c" con igual raíz, sólo se verán las declaraciones originales, pero al incluirlo en un módulo de diferente raíz, sólo se verán las declaraciones con el prefijo *extern*. Para lograrlo, cada fichero fuente debe identificarse a si mismo dentro de su fichero *include*, colocando un #define a comienzo del mismo, y utilizando las directivas del preprocesador #ifdef, #else, y #endif. De esta forma cada fichero que haga referencia a un elemento situado en otro fichero, deberá añadir al principio del mismo un #include del fichero ".h" referenciado. Por otro lado al usar las utilidades *make*, o los modernos gestores de proyectos, cada vez que se haga un cambio en un fichero ".h", provocará que los ficheros que lo incluyen se recompilen de forma automática.

En la mayoría de los compiladores C, esta característica será de gran ayuda en el desarrollo de programas. Así por ejemplo, se puede cambiar fácilmente el modelo de memoria utilizado en un fichero ".h", y conseguir que el cambio se propague a lo largo de todo el proyecto. Veamos a continuación como poner en práctica todo esto:

```
/* Módulo principal - MAIN.C */
#define _MAIN_
/* Define el nombre del módulo para control de la inclusión */
#include <reg517.h>     // Definiciones para la CPU
#include "v24ini537.h"  // Referencias externas a V24INI.C
#include "t0ini537.h"   // Referencias externas a T0INI537.C
#include "rlt_int.h"    // Referencias externas a RLT_INT.C
#include "main.h"       // Referencias locales a MAIN.C
```

```
void main(void) {
   v24ini_537() ;
   timer0_init_537() ;
   while(1) {
      printf("Tiempo = %d:%d:%d:%d",clock.hours,
                                    clock.mins,
                                    clock.secs,
                                    clock.msecs) ;
      }
   if((P1 & 0x01) != 0) {
      clock_run_fl = 1 ; // Si se ha pulsado start clock
      }
   else {
      clock_run_fl = 0 ; // Si se ha soltado stop clock
      }
   if((P1 & 0x02) != 0) {
      clock_reset_fl = 1 ; // Si se ha pulsado clear clock
      }
   }

/* Módulo con la rutina de atención al Timer0 */ /* RLT_INT.C */
#define _RLT_INT_  /* Identifica el nombre del módulo */
/* Referencias externas */
#include "rlt_int.h"    // Referencias locales RLT_INT.C


/***  ATENCION A LA INTERRUPCION DEL TIMER 0  ***/
   void timer0_int(void) interrupt 1 using 1 {
     if(clock.msec++ == 1000) {
        clock.sec++ ;
        if(clock.sec == 60) {
           clock sec = 0 ;
           }
        }
     }

Los ficheros include deben ser:

/* Fichero Include para RLT_INT.C */
/* Estructura para el reloj - Disponible a todos los módulos */

struct time { UCHAR hours ;
              UCHAR mins ;
              UCHAR secs ;
              UCHAR msec  ; } ;

#ifdef _RLT_INT_
/* Declaraciones originales - activas sólo en el módulo raíz */
/* Creación de la estructura en XDATA */
struct time xdata clock ;
bit clock_reset_fl        // Flag para indicar a la interrupción
                          // del timer 0, que ponga el reloj a cero
#else
/* Referencias externas - para módulos distintos al raíz */
extern struct time xdata clock ;
extern bit clock_reset_fl // Flag para indicar a la interrupción
                          // del timer 0, que ponga el reloj a cero
#endif

/* Fichero Include para MAIN.C */
#ifdef _MAIN_
```

```
/* Declaraciones de variables locales */
bit clock_run_fl  ;     // Flag para indicar a la interrupción
                        // del timer 0 que detenga el reloj
#else
/* Referencias externas - para módulos distintos al raíz */
extern struct time xdata clock ;
extern bit clock_run_fl = 0 ;  // Flag para indicar a la interrupción
                        // del timer 0 que detenga el reloj
#endif
/* Fichero Include para V24INI537.C */
#ifdef _V24INI537_
/* Prototipos de funciones originales - para usar en V24INI537.C */
void v24ini_537(void) ;
#else
/* Referencias externas - para usar en otros módulos */
extern void v24ini_537(void) ;
#endif
```

Ahora, si se añade un nuevo dato global, por ejemplo a RLT_INT.C, solo se necesita incluir la declaración original por encima del "#else", y la versión *extern* debajo, lo cual hace que el nuevo dato esté disponible a cualquier módulo que lo necesite. Para resumir, el formato del módulo fuente es:

```
#define _MODULO_
#include <mod1.h>
#include "modulo.h"
.
.
.
funciones()

El formato del fichero include es:

/* Definiciones generales, tales como estructuras y uniones */
#ifdef _MODULO_
/* Poner aquí los prototipos originales y las declaraciones globales
*/
#else
/* Poner aquí referencias externas a las funciones y datos anteriores
*/
#endif
```

Estructura estándar de los módulos en C51

Para ayudar en la integración de este método de construcción de programas, se presentan seguidamente las plantillas estándar para el módulo fuente y para el módulo *include* asociado:

Plantilla para el módulo fuente estándar

```
#define __STD__             /* Define el nombre del módulo */
/*******************************************************/
/* Projecto:       X                                   */
/* Autor:          X          Fecha creación:  XX\XX\XX   */
/* Nombre fichero: X          Lenguaje:        X          */
/* Derechos:       X          Derechos:        X          */
/*                                                     */
/* Compilador:   X               Ensamblador:  X         */
/* Versión:    X.XX              Versión:    X.XX        */
```

```
/*********************************************************/
/* Detalles del módulo:                                  */
/*********************************************************/
/* Propósito:                                            */
/*                                                       */
/*                                                       */
/*********************************************************/
/* Historia de modificaciones                            */
/*********************************************************/
/* Nombre:         X                   Fecha:  XX\XX\XX  */
/* Modificación:   X                                     */
/*                                                       */
/* Nombre:         X                   Fecha:  XX\XX\XX  */
/* Modificación:   X                                     */
/*                                                       */
/* Nombre:         X                   Fecha:  XX\XX\XX  */
/* Modificación:   X                                     */
/*                                                       */
/*********************************************************/
/* Prototipos de Funciones Externas                      */
/*********************************************************/
#include ".h"
/* Ficheros header del Estándar ANSI C                   */
/*********************************************************/
/* Declaraciones de Datos Globales                       */
/*********************************************************/
#include ".h"
/* Fichero header de este módulo                         */
/*********************************************************/
/* Declaracones Externas                                 */
/*********************************************************/
#include  ".h"
/* Ficheros header de otros módulos                      */
/*********************************************************/
/* Detalles de Funciones:                                */
/*********************************************************/
/* Nombre de Función:                                    */
/* Llamada desde:                                        */
/* Llama a:                                              */
/*********************************************************/
/* Propósito: lazo main para programa de entrenamiento   */
/*                                                       */
/*********************************************************/
/* Uso de Recursos:                                      */
/*                                                       */
/* CODE      CONST      DATA       IDATA      PDATA      */
/* n/a        n/a        n/a        n/a        n/a       */
/*                                                       */
/* Prestaciones:                                         */
/* Tiempo Máx Ejecución:      Tiempo Mín Ejecución:      */
/*                                                       */
/*                                                       */
/*********************************************************/
/* Funciones Ejecutables                                 */
/*********************************************************/
/* Fin de STD.c                                          */
/*********************************************************/
```

Plantilla para el módulo "Include" estándar

```
/*********************************************************/
/* Projecto:       X                                     */
/* Autor:          X          Fecha creación:  XX\XX\XX   */
/* Nombre fichero: X          Lenguaje:        X          */
/* Derechos:       X          Derechos:        X          */
/*                                                       */
/* Compilador:   X               Ensamblador:  X         */
/* Versión:    X.XX              Versión:    X.XX        */
/*********************************************************/
/* Historia de modificaciones                            */
/*********************************************************/
/* Nombre:         X                    Fecha:  XX\XX\XX  */
/* Modificación:   X                                      */
/*                                                       */
/* Nombre:         X                    Fecha:  XX\XX\XX  */
/* Modificación:   X                                      */
/*                                                       */
/* Nombre:         X                    Fecha:  XX\XX\XX  */
/* Modificación:   X                                      */
/*                                                       */
/*********************************************************/
/* Definiciones Globales                                 */
/*********************************************************/
/* Estructuras, uniones y otras definiciones             */

#ifdef _STD_
/* Para comprobar la inclusión en el módulo raíz         */
/*********************************************************/
/* Prototipos de Funciones Propias                       */
/*********************************************************/
/* Prototipos de Funciones del Módulo Raíz               */
/*********************************************************/
/* Declaraciones de Datos Propias                        */
/*********************************************************/
/* Declaraciones de Datos del Módulo Raíz                */
/*********************************************************/
#else
/*********************************************************/
/* Prototipos de Funciones Externas                      */
/*********************************************************/
/* Prototipos de Funciones Externas para otros Módulos   */
/*********************************************************/
/* Declaraciones de Datos Externas                       */
/*********************************************************/
/* Declaraciones de Datos Externas para otros Módulos    */
/*********************************************************/
#endif
```

Resumen

Se puede reducir el tiempo de edición de un proyecto, si a cada nuevo módulo se le añade en su primera línea el *define* correspondiente, y si los nuevos objetos globales se colocan en su fichero ".h" asociado. Además se reduce el tiempo de compilación y los errores de linkado. Las definiciones de las estructuras y uniones sólo aparecen una vez, con lo que se eliminan problemas de compilación y linkado.

4.4.- REPARTO DE TAREAS

4.4.1.- Aplicaciones con el 8051

Muchas personas han empezado a programar utilizando el lenguaje BASIC en un PC o máquina similar. Los programas que realizan inicialmente no suelen ser muy complicados. Empiezan a correr cuando se teclea "RUN" y terminan en un END o STOP. Mientras tanto, el PC se dedica totalmente a la ejecución del típico programa "HELLO WORLD". Cuando el programa termina, se retorna al editor BASIC, o al entorno de trabajo utilizado.

La experiencia es muy buena y el nuevo programador cree que ya sabe programar. Sin embargo, cuando se escribe un programa para un microcontrolador como el 8051, el problema del comienzo y final del programa se presenta rápidamente. Típicamente, el software de un sistema basado en el 8051 está formado por múltiples programas individuales, que ejecutados conjuntamente, contribuyen a la consecución del objetivo final. Entonces, un problema fundamental es asegurarse de que todas las partes del programa se ejecutan.

4.4.2.- Sistemas sencillos con el 8051

El enfoque más sencillo es llamar a cada una de las sub-funciones del programa de una forma secuencial, de tal forma que después de un cierto tiempo, cada parte se habrá ejecutado el mismo número de veces. Esto constituye el bucle de fondo (*background loop*), o programa principal. En primer plano (*foreground*) aparecen las funciones de interrupción, iniciadas por sucesos producidos en tiempo real, tales como señales de entrada o desbordamiento de temporizadores.

El intercambio de datos entre las interrupciones y el programa principal se realiza usualmente a través de variables globales y *flags*. Este tipo de programas puede funcionar correctamente si se tiene cuidado en el orden y frecuencia de ejecución de cada sección.

Las funciones llamadas por el programa principal deben escribirse de tal forma que en cada llamada se ejecute una sección particular de su código. Así al entrar en esta función, se toma la decisión de la tarea que le toca ejecutar, se ejecuta esa parte y se sale de la función, cambiando posiblemente algunos *flags* que indicarán la tarea a realizar en la siguiente llamada. De esta forma, cada bloque funcional debe mantener su propio sistema de control que asegure la ejecución del código adecuado en cada entrada al mismo.

En un sistema de este tipo, todos los bloques funcionales tienen la misma importancia, y no se entra en un nuevo bloque hasta que no le toque su turno dentro del programa principal. Sólo las rutinas de interrupción pueden romper este orden, aunque también están sujetas a un sistema de prioridades. Si un bloque necesita una cierta señal de entrada, o se queda a la espera de la misma impidiendo que otros bloques puedan ejecutarse, o cede el control hasta que le vuelva a tocar su turno dentro del bucle principal. En este último caso se corre el riesgo de que el suceso esperado se produzca y no tenga una respuesta, o que la respuesta llegue demasiado tarde. No obstante, los sistemas de este tipo funcionan bien en programas en las que no hay secciones que tengan exigencias críticas en los tiempos de respuesta.

Los llamados sistemas en tiempo real no son de este tipo. Típicamente contienen código cuya ejecución origina o es originada por sucesos del mundo real, que trabajan con datos procedentes de otras partes del sistema, cuyas entradas pueden cambiar lenta o rápidamente.

El código que contribuye en mayor medida a la funcionalidad del sistema, debe tener precedencia sobre las secciones cuyos objetivos no son críticos respecto al objetivo final. Sin embargo, muchas de las aplicaciones con el 8051 tienen partes con grandes exigencias de tiempo de respuesta, que suelen estar asociadas a interrupciones. La necesidad de atender a las interrupciones tan rápido como sea posible, requiere que el tiempo de ejecución de las mismas sea muy corto, ya que el sistema dejará de funcionar si el tiempo de respuesta a cada interrupción supera al intervalo de tiempo en la que las interrupciones se producen.

Por regla general se consigue un nivel aceptable de prestaciones si las funciones complejas se llevan al programa principal, y se reservan las interrupciones para las secciones con exigencias de tiempo críticas. En cualquier caso, aparece el problema de la comunicación entre el programa principal y las rutinas de interrupción.

Este sencillo sistema de reparto de tareas trata a todas ellas de igual forma. Cuando la CPU se encuentra muy cargada de trabajo por tener que atender a entradas que cambian con rapidez, puede ocurrir que el programa principal no corra con la suficiente frecuencia y la respuesta transitoria del mismo se degrade.

4.4.3.- Reparto de tareas simple - Una solución parcial

Los problemas de los sistemas de bucle simple pueden solucionarse parcialmente, controlando el orden y la frecuencia de las llamadas a funciones. Una posible solución pasa por asignar una prioridad a cada función y establecer un mecanismo que permita a cada función especificar la siguiente función a ejecutar. Las funciones de interrupción no deben seguir este orden y deben ser atendidas con rapidez. Los sistemas de este tipo pueden resultar útiles, si el tiempo de ejecución de las funciones no es excesivo.

Una solución alternativa consiste en utilizar un temporizador, que asigne un tiempo de ejecución a cada trabajo a realizar. Cada vez que el tiempo asignado se supere, la tarea en curso se suspende y comienza otra tarea.

Desafortunadamente todas estas posibilidades suelen intentarse tarde, cuando los tiempos de respuesta se alargan en exceso. En estos casos, lo que había sido un programa bien estructurado termina degenerando en código spaghetti, plagado de ajustes y modos especiales, tendentes a superar las diferencias entre las demandas de los sucesos del mundo real, y la respuesta del sistema. En muchos casos, los mecanismos de control de las funciones llamadas generan una sobrecarga que acentúa aún más el desfase entre las exigencias y la respuesta.

La realidad es que los sucesos del mundo real no siguen ningún orden ni pueden predecirse. Algunos trabajos son más importantes que otros, pero el mundo real produce sucesos a los que hay que responder inmediatamente.

4.4.4.- Un enfoque práctico

Si no se recurre a un sistema operativo en tiempo real como RTX51, ¿Qué se puede hacer?

Un mecanismo sencillo para controlar el programa principal, puede ser una sentencia *switch*, con la variable *switch* controlada por algún suceso externo en tiempo real. Idealmente, éste debe ser la rutina de interrupción de mayor prioridad. Las tareas del programa principal con mayor prioridad se colocan en los *case* de menor valor numérico, y las de menor prioridad en

los *case* con mayor valor numérico. Cada vez que se ejecuta una tarea, se incrementa la variable *switch*, permitiendo que se ejecuten las tareas de menor prioridad. Si se produce la interrupción, se asigna a la variable *switch* el valor correspondiente a la tarea de mayor prioridad. Pero si la interrupción tarda bastante en producirse nuevamente, la variable *switch* permitirá la ejecución de la tarea de menor prioridad, para después comenzar automáticamente con la de mayor prioridad.

Si la interrupción se produce en el case de nivel 2, la variable *switch* se pone a 0 y así sucesivamente. En este caso las tareas de menor prioridad se ignoran. El sistema no es obviamente ideal, ya que solo la tarea de mayor nivel se ejecuta con la suficiente frecuencia. Sin embargo, bajo condiciones normales puede ser una forma útil de asegurar que las tareas de baja prioridad no se ejecuten con mucha frecuencia. Por ejemplo, no tiene mucho sentido medir la temperatura ambiente más de una vez cada segundo. En un sistemas de este tipo, la tarea encargada de medir la temperatura ambiente estaría situada a nivel 100, en el sistema de reparto de tareas.

Este método falla cuando una tarea de baja prioridad tiene un tiempo de ejecución muy largo. Incluso si se produce la interrupción que exige que el bucle regrese a la tarea de mayor prioridad, el salto a la misma no se producirá mientras no termine la tarea en curso. Para que se produzca la situación deseada se necesita de un mecanismo de rebanado de tiempo (*time-slice*).

Un truco útil consiste en utilizar una interrupción libre para garantizar que la tarea de alta prioridad se ejecute a tiempo. Para ello se asigna la tarea de alta prioridad a la rutina de servicio de la interrupción libre o sobrante. Así cuando se produzca la interrupción en tiempo real, y justo antes de salir de la rutina de servicio de la misma, se pondrá a uno el *flag* de petición de la interrupción libre, con lo cual, tras el RETI comenzará la ejecución de la tarea de alta prioridad. Por supuesto, la interrupción libre debe ser de baja prioridad.

Hay que tener en cuenta que en estos casos el factor más importante es conseguir el menor tiempo de ejecución posible, y particularmente en las rutinas de interrupción. Ello significa que se debe hacer un uso completo de las extensiones de C51, tales como los punteros específicos, los bits de funciones especiales y las variables locales de tipo *register*.

Capitulo 5.- Extensiones al lenguaje C para el 8051.

5.1.- INTRODUCCIÓN

La programación del 8051 trata principalmente con accesos a dispositivos reales en direcciones específicas, además de proporcionar servicio a las interrupciones. C51 dispone de muchas extensiones al lenguaje C, con el fin de conseguir una eficiencia del código próxima a la del lenguaje ensamblador del 8051. Los aspectos centrales de estas extensiones se tratan seguidamente.

5.2.- ACCESO A LOS PERIFERICOS INTERNOS DEL 8051

En las aplicaciones de control típicas con mucha frecuencia se producen operaciones tales como: lectura y escritura de datos en un puerto, asignación de valor a los registros de un *timer*, lectura de valores de registros captura, etc. El compilador C51 dispone de tipos de datos especiales como **sfr** y **sbit**, para realizar estas operaciones sin necesidad de acudir al lenguaje ensamblador.

Algunas declaraciones típicas son:

```
sfr P0 = 0x80;
sfr P1 = 0x90;
sfr ADCON = 0xC5;
sbit EA = 0xAF;
```

Muchas de estas declaraciones residen en ficheros cabecera tales como reg51.h en el caso del 8051, o en reg552.h para el 80C552. Son precisamente estas declaraciones las que adaptan el compilador a un derivado del 8051. Con ellas, el acceso a los sfr resulta muy sencillo:

```
{
ADCON = 0x08 ;   /* Escritura de un dato en un registro */
P1 = 0xFF    ;   /* Escritura de un dato en un puerto */
io_estado = P0 ; /* Lectura del estado de un puerto */
EA = 1       ;   /* Poner a 1 un bit (enable all interrupts) */
}
```

No todos los registros especiales de un microcontrolador son direccionables a nivel bit. La regla es que sólo los sfr cuya dirección es divisible por 8 son direccionables a nivel bit. Si ello no sucede, el acceso a un bit debe hacerse mediante instrucciones de byte y utilizando máscaras.

Consultar el manual de usuario del procesador para verificar si un sfr es direccionable a nivel de bit.

5.3.- INTERRUPCIONES

Las interrupciones juegan un papel importante en muchas aplicaciones del 8051. Cuando se atiende a una interrupción se debe tener en cuenta diversos factores:

- Para cada interrupción debe generarse un vector, que dirija al procesador a la rutina de servicio apropiada. El compilador C51 realiza este trabajo de forma automática.

- Las variables locales de la rutina de servicio de la interrupción no deben compartirse con las variables locales del programa principal.
- Las rutinas de servicio a las interrupciones no deben modificar ningún registro ni variable que sea utilizado por el programa principal.
- Las rutinas de servicio a las interrupciones deben ser lo más cortas posibles.

5.3.1.- Funciones de interrupción

Para definir una función de interrupción en C, se utiliza un tipo especial *interrupt*:

```
timer0_int() interrupt 1 using 2
{
unsigned char temp1;
unsigned char temp2;
siguen las sentencias C ejecutables;
}
```

En primer lugar, para cada función de interrupción se genera un vector en la dirección de memoria de código (8*n+3), donde n es el argumento de *interrupt* (n=1 en el ejemplo anterior). En este caso en la dirección 0BH de memoria de código, el compilador C51 coloca un "LJMP timer0_int". Además el linker se ocupa de no solapar las variables locales de la función de interrupción, con las variables locales del programa principal.

Las funciones de interrupción no son llamadas por el usuario, se llaman de forma automática cuando se produce la interrupción. Por ello estas funciones no reciben parámetros ni devuelven valores. El compilador se encarga de guardar en la pila los registros que utilice la función de interrupción, en cuanto se entra en ella. También se ocupa de sacar de la pila los valores guardados, justo antes de salir de la función. Además C51 hace que las funciones de interrupción terminen en un RETI, en lugar del RET con el que terminan las funciones normales.

5.3.2.- Uso del compilador C51 con tarjetas provistas de monitores&depuradores

Muchas tarjetas 8051, van provistas de un programa monitor&depurador en EPROM situado en dirección 0 del área CODE, con una RAM en dirección 0x8000 visible en las áreas CODE y XDATA gracias al producto de las señales PSEN y RD del μC. Esta situación tan común, presenta problemas con los vectores de interrupción que ocupan las direcciones: 3, 0Bh, 13H, 1BH, 23H, ...situadas en la EPROM. La mayoría de los programas monitor de estas tarjetas solucionan el problema redireccionando todos los vectores por encima de la dirección 0x8000 (hacia la RAM), para lo cual le añaden un desplazamiento de 0x8000 a cada vector. Así la rutina de atención al desbordamiento del *timer* 0 se trata en la dirección 0x800B en lugar de hacerlo en la dirección 0x0B.

Para realizar lo anterior, hasta la versión 3.40 de C51, la solución consistía en deshabilitar la generación automática de los vectores de interrupción, y escribir los vectores deseados en lenguaje ensamblador. Sin embargo desde la versión 3.40 se dispone del control INTVECTOR que facilita la tarea, ya que los vectores de interrupción se dirigen a las direcciones:

8 * n + 3 + INTVECTOR

Para utilizar este control se usa la directiva #pragma del preprocesador:

```
#pragma INTVECTOR(0x8000)/*Dirección inicial para los vectores =
0x8000*/

void timer0_int(void) interrupt 1 {

   /* Código...*/

   }
```

Lo cual produce un LJMP timer0_int en la dirección C:0x800B. Esta instrucción junto al salto que el programa monitor incluye desde la dirección C:0x0B hacia la dirección C:0x800B hace que las interrupciones funcionen correctamente.

5.3.3.- Espaciado de Interrupciones distinto que 8

Algunos derivados del 8051 no mantienen el espaciado de 8 bytes entre los vectores de interrupción, haciendo inservible la fórmula (8 * n + 3). Afortunadamente, el "#pragma INTERVAL(n)" soluciona esta situación, ya que los vectores de interrupción se dirigen a las direcciones:

INTERVAL * n + 3 + INTVECTOR

```
#pragma INTERVAL(3)   /* Cambia el espaciado a 3 bytes */
```

Por omisión los valores utilizados son: INTERVAL(8) e INTVECTOR(0)

5.3.4.- El control using

El control *using n* ordena al compilador que utilice el banco de registros n. El compilador C51 hace un uso intensivo de los registros R0...R7 del 8051. Normalmente se utiliza el banco de registros 0. Sin embargo, como puede apreciarse en la sección 5.2.1. se puede hacer que una función de interrupción utilice otro banco de registros diferente al utilizado por el programa principal. De esta forma el tiempo utilizado por las interrupciones se reduce, ya que no necesita empilar y desempilar los registros cada vez que se atienden.

Dos interrupciones de la misma prioridad pueden utilizar el mismo banco de registros debido a que no es posible que una de ellas interrumpa a la otra.

Si el tiempo de ejecución de las funciones de interrupción no es crítico, puede omitirse el uso del control *using*, en este caso C51 examina los registros que utiliza la interrupción, los guarda en la pila al entrar y los saca de la pila al salir. Esto lógicamente aumenta el tiempo utilizado por las funciones de interrupción.

5.4.- INTERRUPTS, USING, REGISTERBANKS, NOAREGS EN C51

Las interrupciones juegan un papel importante en la mayoría de las aplicaciones del 8051, y afortunadamente C51 permite escribir las funciones de interrupción enteramente en C. Aunque es perfectamente posible escribir código que funcione, respetando el estándar ANSI C, si se quiere mejorar la eficacia del código es conveniente comprender la utilidad de los siguientes controles:

- INTERRUPT
- USING
- NOAREGS
- RE-ENTRANT
- REGISTERBANK

5.4.1.- El atributo básico de las funciones de interrupción

Para que una función de interrupción pueda ser alcanzada es necesario generar el vector de interrupción adecuado. El compilador C51 lo hace de forma automática, basándose en el argumento de la palabra *interrupt*. Posteriormente, el linker impide que las variables locales de las funciones de interrupción se solapen con las del programa principal, creando secciones especiales en RAM. Ejemplo:

```
/* Rutina de atención a la interrupción por desbordamiento del Timer
0*/

timer0_int() interrupt 1
{
unsigned char temp1 ;
unsigned char temp2 ;

/* Sentencias C ejecutables ; */
}
```

- La palabra *interrupt n* genera un vector de interrupción en la dirección (8*n+3). En este caso se coloca un "LJMP timer0_int" en la dirección 0BH de memoria de código.
- Las variables locales declaradas en la función no se solapan con las variables del programa principal (trabajo del linker).
- El compilador añade el código necesario para empilar los registros **utilizados** (ACC, B, DPTR, PSW, R0-R7) a comienzo de la función.
- Al salir de la función se desempilan los registros anteriores y se inserta un RETI en lugar de un RET.

Tomando como ejemplo la función de atención a la interrupción por desbordamiento del *timer* 0, y suponiendo que esta no utilice registros:

Código de entrada a la función timer0_int

```
void timer0_int(void) interrupt 1
{
RSEG ?PR?timer0_int?TIMER0
USING 0
timer0_int:
; LINEA DE CODIGO FUENTE # 2
```

Si la función de interrupción llama a su vez a otra función llamada "sys_interp", el código de entrada cambia a:

Código de entrada a la función timer0_int que llama a otra función

```
; void timer0_int(void) interrupt 1
{
RSEG ?PR?timer0_int?TIMER0
USING 0
timer0_int:
PUSH ACC
PUSH B
PUSH DPH
PUSH DPL
PUSH PSW
PUSH AR0
PUSH AR1
PUSH AR2
PUSH AR3
PUSH AR4
PUSH AR5
PUSH AR6
PUSH AR7
```

Ahora, al entrar a la función de interrupción se guardan en la pila todos los registros, ya que C51 supone que la función llamada (sys_interp) puede hacer uso de los mismos. Si se observa la entrada a sys_interp, se hace evidente un truco importante del compilador:

Código de entrada a sys_interp()

```
; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
 USING 0
_sys_interp:
MOV y_value?10,R5
MOV map_base?10,R2
MOV map_base?10+01H,R3
;--Variable 'x_value?10' assigned to Register 'R1' --
MOV R1,AR7
```

Puede observarse la forma tan eficiente para mover el contenido de R7 a R1, utilizando AR7. Este tipo de direccionamiento absoluto de registros proporciona un código muy rápido.

5.4.2.- El truco del direccionamiento absoluto de registros en detalle

El 8051 no dispone de la instrucción MOV Reg,Reg, por ello Keil utiliza el truco de considerar un registro como una dirección *data* absoluta:

Simulación de la instrucción MOV Reg,Reg:

En el banco de registros 0 - MOV R0,AR7, es idéntica a - MOV R0,07H.

Este truco solo puede utilizarse cuando el compilador conoce el banco de registros que se está utilizando. Cuando se utiliza el control USING, pueden surgir problemas. Véanse las siguientes secciones...

5.4.3.- El control USING

El control *using n* ordena al compilador que utilice el banco de registros n. De esta forma las rutinas de interrupción con exigencias de tiempo muy críticas pueden cambiar de "contexto", con mayor rapidez que empilando los registros. Además las funciones de interrupción de igual prioridad pueden compartir el mismo banco de registros, al no existir el riesgo de que se interrumpan entre ellas.

5.4.4.- Direcciones de base de los bancos de registros del 8051

Los registros R0...R7 ocupan direcciones consecutivas en la RAM interna del 8051, siendo la dirección base, o dirección correspondiente al registro R0, variable en función del banco de registros que se encuentre activo. Así la dirección base puede ser: 0x00, 0x08, 0x10, o 0x18 según que el banco de registros activo sea el: 0, 1, 2, o 3.

Si a una función de interrupción se le añade el control "USING 1", se sustituye el empilado de los registros por la instrucción "MOV PSW,#08H" que conmuta el banco de registros. El tiempo de entrada a la interrupción disminuye considerablemente, pero puede fallar el direccionamiento absoluto de registros si no se tiene cuidado. Si la función de interrupción no hace uso de registros, y no llama a ninguna otra función, el optimizador elimina el código de banco de registros.

Código de entrada a timer0_int con USING

```
Con USING 1
; void timer0_int(void) interrupt 1 using 1 {
RSEG ?PR?timer0_int?TIMER0
USING 1 <--- Nuevo banco de registros
timer0_int:
PUSH ACC
PUSH B
PUSH DPH
PUSH DPL
PUSH PSW
MOV PSW,#08H
```

Código de entrada a sys_interp()

Usando todavía el banco de registros 0

```
; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
USING 0
_sys_interp:
MOV y_value?10,R5
MOV map_base?10,R2
MOV map_base?10+01H,R3;
--Variable 'x_value?10' assigned to Register 'R1' --
MOV R1,AR7      <----- FALLA!!!!
```

El direccionamiento absoluto de registros supone que el banco de registros activo es el 0, y el programa falla.

5.4.4.- Notas sobre llamadas a funciones desde las interrupciones

C51 utiliza un cierto grado de inteligencia al entrar en las funciones de interrupción. Además de sustituir el RET del final de la función por un RETI, automáticamente empila los registros que utilice la función.

Sin embargo, hay algunos aspectos a considerar:

- Si la función de interrupción llama a cualquier función, C51 empila todos los registros Ri, independientemente de que sea necesario o no. El tiempo utilizado por los 8 PUSH y los 8 POP es de 32 µs a 12 MHz, que puede ser inaceptable en algunos casos. Por lo tanto, o se evita la llamada a funciones o se utiliza el control USING. Puesto que al empilar los 8 registros se utilizan 8 posiciones de RAM interna, y el cambio de banco de registros utiliza también 8 posiciones de RAM interna, no existe diferencia en cuanto al uso de RAM.
- Las variables declaradas dentro de una función de interrupción no deben solaparse con las variables del programa principal, ni con las utilizadas por otras interrupciones.
- Nunca se debe llamar a una función de interrupción desde el programa principal. A veces ello resulta tentador, sobre todo en la fase de inicialización de un programa. Esto puede producir confusión en el linker, con lo que podrían sobre-escribirse las variables del programa principal.
- Si las funciones llamadas por una función de interrupción utilizan menos de 8 registros, el uso del control USING utiliza más RAM que si se empilaran los registros. Ello sin embargo no es razón suficiente para evitar el uso del control USING.
- Las interrupciones de la misma prioridad pueden utilizar el mismo banco de registros, ya que no es posible que se interrumpan entre ellas mismas.

5.4.5.- Cuando usar el control USING

- En interrupciones en las que el tiempo de respuesta es más importante que la cantidad de RAM utilizada.
- En las interrupciones que llaman a otras funciones.

5.4.6.- El #pragma NOAREGS

Direccionamiento absoluto de registros con C51.

Ya se ha comentado que el 8051 no dispone de la instrucción MOV Reg,Reg, por lo que C51 utiliza MOV R1,AR7 donde AR7 es la dirección absoluta del registro R7 en uso. Para que la sustitución funcione correctamente el compilador debe conocer el banco de registros que está utilizando. Si una función se llama desde una interrupción que utiliza el control USING, hay dos posibilidades:

- **N**o usar direccionamiento absoluto de registros, poniendo #pragma NOAREGS antes de la función, y #pragma RESTORE o #pragma AREGS después de la función.

- Indicar a C51 el banco de registros apropiado con #pragma REGISTERBANK(n).

Las funciones compiladas con NOAREGS sirven **siempre**, independientemente del banco de registros que se utilice, aunque pueden ser más lentas que las que utilicen un banco de registros definido.

5.4.7.- El control REGISTERBANK como alternativa a NOAREGS

El control #pragma REGISTERBANK(n) informa a C51 sobre el banco de registros utilizado, haciendo posible el direccionamiento absoluto de registros.

EJEMPLO:

```
/* Rutina de atención a la interrupción por desbordamiento del Timer 0 */
timer0_int() interrupt 1 USING 1 {
unsigned char temp1 ;
unsigned char temp2 ;
/* Sentencias C ejecutables ; */
}
```

Función llamada por timer0_int:

```
#pragma SAVE // Recuerda el banco de registros actual
#pragma REGISTERBANK(1)        // Informa a C51 sobre el nuevo banco de
registros
void sys_interp(char x) {// Función llamada desde interrupción con "using 1"
/* Código */
 }
#pragma RESTORE // Repone el banco de registros original
```

Al utilizar #pragma REGISTERBANK(1) con sys_interp() se restaura el direccionamiento absoluto de registros ya que C51 conoce el banco de registros utilizado.

Nota: Utilícese siempre el control REGISTERBANK(n) para las funciones llamadas desde interrupciones con USING n.

Código de entrada de sys_interp() con REGISTERBANK(n)

```
; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
USING 1
_sys_interp:
MOV y_value?10,R5
MOV map_base?10,R2
MOV map_base?10+01H,R3;--
Variable 'x_value?10' assigned to Register 'R1' --
MOV R1,AR7
```

5.4.8.- Resumen de USING y REGISTERBANK

Expresado en pseudo-código

```
if(interrupt routine = USING 1){
las funciones llamadas desde aquí deben usar #pragma REGISTERBANK(1)
}
```

Nota: las funciones llamadas por la interrupción, sólo pueden llamadas desde funciones que utilicen el banco de registros 1.

5.4.9.- Re-entrancia en C51 - La solución definitiva

A veces una misma función se llama desde interrupciones y desde el programa principal. Para que la llamada desde dos lugares diferentes no tenga consecuencias desastrosas, la función llamada debe ser re-entrante. El usuario puede especificar funciones re-entrantes utilizando el atributo *reentrant* en la definición de las mismas. El aviso del linker "MULTIPLE CALL TO SEGMENT" es el primer signo de que se está tratando de utilizar una función de forma re-entrante.

La razón por la cual una función no re-entrante no puede ser llamada a la vez desde el programa principal y desde una interrupción, es que C51 asigna lugares de almacenamiento en RAM, para las variables locales y para los parámetros de las funciones ordinarias.

El valor asignado a ?C_IBP en el fichero startup.a51, le dice a C51 el lugar donde debe colocar la pila artificial para las funciones re-entrantes. Cada vez que se llama a una función re-entrante, los parámetros de entrada a la misma se llevan desde los registros al área de RAM que comienza en la dirección indicada por ?C_IBP. De igual forma, cualquier variable local utilizada por una función re-entrante se coloca en esta pila especial.

Cuando, antes de main( ), se ejecuta startup.a51, la línea:

```
IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)
MOV ?C_IBP,#LOW IBPSTACKTOP
ENDIF
```

inicializa ?C_IBP al valor que se le haya asignado anteriormente a IBPSTACKTOP. A medida que se "empilan" variables locales en la pila re-entrante, se decrementa ?C_IBP. Así, si se produce una interrupción que llama a la función de nuevo, las variables locales se guardan en el valor actual de ?C_IBP, sin destruir los valores anteriores.

Obtención de una variable local con offset 2 desde el valor actual de la pila re-entrante:

```
MOV R0,?C_IBP ;Obtener la base de la pila re-entrante
MOV A,@R0
ADD A,#002    ; Añadir el offset
MOV A,@R0     ; Obtener la variable local por direccionamiento
indirecto
MOV R7,A      ;Guardar su valor
...
```

?C_IBP actúa como un puntero a la base de la pila re-entrante, utilizado para acceder a las variables locales de las funciones re-entrantes. Al salir de la función se restaura el valor de

?C_IBP a su valor inicial, sumando el tamaño de las variables locales y parámetros utilizados. Esto representa una sobrecarga de trabajo, que indica que la re-entrancia solo debe utilizarse cuando sea absolutamente necesaria.

Si se añade el atributo *reentrant* a la función sys_interp(), todavía se necesita el control NOAREGS ya que se ha cambiado el banco de registros con USING 1. De echo, cualquier función re-entrante debe llevar el atributo NOAREG para hacerla totalmente independiente del banco de registros.

Código de entrada de sys_interp()

```
; unsigned char interp_sub(unsigned char x,
RSEG ?PR?_?interp_sub?INTERP
USING 0
_?interp_sub:
DEC ?C_IBP
DEC ?C_IBP
MOV R0,?C_IBP
XCH A,@R0
MOV A,R2
XCH A,@R0
INC R0
XCH A,@R0
MOV A,R3
XCH A,@R0
DEC ?C_IBP
MOV R0,?C_IBP
XCH A,@R0
MOV A,R5
XCH A,@R0
DEC ?C_IBP
MOV R0,?C_IBP
XCH A,@R0
MOV A,R7
XCH A,@R0
DEC ?C_IBP ;
```

Código de salida de sys_interp()

```
?C0009:
MOV A,?C_IBP
 ADD A,#010H   <-- Restaura ?C_IBP a su valor original
position
MOV ?C_IBP,A
RET ;
END OF _?sys_interp
```

5.4.10.- Resumen de controles para funciones de interrupción

Utilizando las siguientes combinaciones de controles se evitan los avisos del linker y el código potencialmente peligroso.

Atributo de función de Interrupc.	Atributo de la función llamada:
	"no re-entrante"
No USING	no requiere ningún atributo
USING n	USING n o #pragma REGISTERBANK(n) o #pragma NOAREGS

5.4.11.- Re-entrancia y funciones de librería

La mayoría de las funciones de libreria de C51 son re-entrantes y pueden utilizarse libremente desde el programa principal y desde las interrupciones. Sin embargo, algunas de las funciones de mayor tamaño, tales como printf(), scanf() etc. no son re-entrantes. Si se utiliza una función no re-entrante de manera re-entrante, se obtiene el aviso "MULTIPLE CALL TO SEGMENT", como cabía esperar.

Las funciones de librería "ocultas" que se encargan de las operaciones con enteros (multiplicación, división, etc.) son todas re-entrantes, lo cual hace que se puedan realizar libremente divisiones de 16 bits en las rutinas de interrupción o en el programa principal.

En cualquier caso, cuando se utilicen funciones de librería de forma re-entrante, es conveniente consultar el manual de C51 para obtener detalles del carácter de esas funciones.

Capítulo 6 - Punteros en C51

6.1.- INTRODUCCIÓN

Aunque los punteros pueden utilizarse igual que en los programas C para PC, en C51 hay varias extensiones importantes dirigidas a lograr un código más eficiente.

6.2.- USO DE PUNTEROS Y ARRAYS EN C51

Los punteros constituyen paradójicamente uno de los puntos fuertes y a la vez puntos débiles del lenguaje C. El uso, o más apropiadamente el abuso de los punteros es la razón por la que algunos consideran peligroso al lenguaje C.

6.2.1.- Los Punteros en Ensamblador

Para un programador en ensamblador, los punteros de C equivalen al direccionamiento indirecto que realizan las siguientes instrucciones del 8051:

```
MOV  R0,#40
MOV  A,@RO            ; Lectura de RAM interna

MOV  R0,#40
MOVX A,@RO            ; Lectura de RAM externa paginada

MOVX A,@DPTR          ; Lectura de RAM externa

CLR  A
MOV  DPTR,#0040
MOVC A,@A+DPTR        ; Lectura de memoria de código
```

En todos los casos anteriores el dato se encuentra en una posición de memoria cuya dirección se encuentra en los registros situados a la derecha de '@'.

6.2.2.- Los Punteros en C51

El puntero es el equivalente en C al acceso indirecto del ensamblador. Puede definirse el puntero *como una variable que contiene la dirección de otra variable o constante.*

Se define un puntero a unsigned char así:

unsigned char *puc;

*Notar que el prefijo *, indica que el dato residente en puc es una dirección y no un valor.*

En todos los ejemplos anteriores en ensamblador se utilizan dos operaciones:

1. Poner en un registro la dirección de la variable a la que se desea acceder.
2. Usar la instrucción apropiada para acceder indirectamente al dato.

En C se utiliza el mismo procedimiento, aunque se debe definir el puntero, mientras que en ensamblador se utilizan los registros del 8051.

```
/* 1 - Definir un puntero que almacena dirección de otra variable */

unsigned char *puc ;

/* 2 - Poner en el puntero la dirección de la variable a la que se */
     /* desea acceder indirectamente */

puc = &variable_c ;

/* 3 - Poner el dato '0xff' indirectamente en variable_c */

*puc = 0xff ;
```

Los pasos anteriores hacen:

1. Reserva espacio en RAM para un puntero. El compilador asigna un nombre simbólico a una posición de RAM, como hace con una variable normal.
2. Carga el espacio reservado en RAM con la dirección de la variable a la que se desea acceder, equivale a 'MOV R0,#40'. La sentencia "puc = &variable_c" significa poner en puc la dirección de variable_c. En este caso el puntero se corresponde con R0 y '&' equivale al símbolo '#' utilizado en ensamblador.
3. Mover el dato 0xff a la variable cuya dirección se encuentra en puc, lo que equivale a la instrucción 'MOV A,@R0' del ejemplo en ensamblador.

La posibilidad de asignar valores de forma directa, x = valor, o de forma indirecta, x = *y_ptr, es extremadamente útil. Veamos un ejemplo en C:

```
/* Demostración del uso de Punteros */

char c1;        // 1 - Define la variable c1, de tipo char
char *pc;       // 2 - Define un puntero a variable de tipo char
                       que todavía no apunta a ningún lugar
main() {
   c1 = 0xff;   // 3 - Asigna (directamente) el valor 0xff a c1
   pc = &c1;    // 4 - Asigna a pc la dirección de c1
   *pc = 0xff;  // 5 - Asigna (indirectamente) el valor 0xff a c1
   }
```

Nota: La línea 4 hace que pc apunte a la variable c1. Una forma alternativa de hacer esto es:

```
/* Demostración del uso de Punteros */

char c1;        // 1 - Define la variable c1, de tipo char
char *pc = &c1; // 2 - Define un puntero a variable de tipo char
                       que apunta a la variable c1
main() {
   c1 = 0xff;   // 3 - Asigna (directamente) el valor 0xff a c1
   *pc = 0xff;  // 5 - Asigna (indirectamente) el valor 0xff a c1
   }
```

Los punteros con el prefijo *, pueden utilizarse igual que las variables normales. La sentencia:

```
x = y + 3 ;
```

puede realizarse igualmente con punteros:

```
char x, y ;
char *x_ptr = &x ;
char *y_ptr = &y ;
*x_ptr = *y_ptr + 3 ;
```

o:

```
x = y * 25 ;
*x_ptr = *y_ptr * 25 ;
```

La cosa más importante a recordar con punteros es que

```
*ptr = var ;
```

significa "asignar el valor var a la variable apuntada por ptr", mientras que

```
ptr = &var ;
```

significa "asignar a ptr la dirección de la variable var".

6.3.- PUNTEROS A DIRECCIONES ABSOLUTAS

En muchas aplicaciones con el 8051, la ROM, la RAM y los periféricos se encuentran en direcciones fijas. En consecuencia, se plantea la necesidad de hacer que un puntero apunte a una dirección absoluta, en lugar de apuntar a variables cuya dirección de memoria es desconocida y la mayoría de las veces irrelevante.

El método más sencillo es asignar la dirección durante la compilación:

```
char *abs_ptr = 0x8000 ;  // Define un puntero a char que apunta
                          // a la dirección 0x8000
```

Sin embargo si la dirección a donde debe apuntar el puntero no es conocida durante la compilación, pero si durante la ejecución del programa, el procedimiento anterior no sirve. En este caso se declara un puntero, y más tarde se le asigna una dirección:

```
char *abs_ptr ;  // Declara un puntero

abs_ptr = (char *) 0x8000 ;  // Apunta a la dirección 0x8000
*abs_ptr = 0xff ;            // Escribe 0xff en dirección 0x8000
*abs_ptr++ ;                 // Apunta a la dirección 0x8001
```

6.4.- ARRAYS Y PUNTEROS - ¿DOS CARAS DE LA MISMA MONEDA?

6.4.1.- Arrays no inicializados

Las variables declaradas vía:

```
unsigned char x ;
unsigned char y ;
```

son posiciones de memoria de 8 bits. Las declaraciones:

```
unsigned int a ;
unsigned int b ;
```

reservan cuatro posiciones de memoria dos para 'a', y dos para 'b'. En varios lenguajes de programación es posible agrupar variables del mismo tipo en arrays. En BASIC se crea un array con DIM a(10).

De igual forma en 'C' se declara un array, así:

unsigned char a[10] ;

Su efecto es reservar diez posiciones de memoria consecutivas, comenzando en la dirección de 'a'. Como no hay nada al lado derecho de la definición, no se asignan valores iniciales a los elementos del array.

6.4.2.- Inicialización de Arrays

Una situación más usual con arrays es:

unsigned char test_array [] = { 0x00,0x40,0x80,0xC0,0xFF } ;

en la que los valores iniciales se asignan antes de que el control del programa llegue a "main()". Notar que no es necesario poner entre corchetes el tamaño del array, ya que el compilador lo ajusta automáticamente.

Otro ejemplo corriente en BASIC es el de una cadena, o array de caracteres como:

```
A$ = "HELLO!"
```

En C se escribe así:

```
char test_array[] = "HELLO!";
```

En C no hay distinción entre cadenas y arrays, ya que en C un array es un conjunto secuencial de bytes ocupados por caracteres o números. De echo, el campo de los apuntadores y los arrays se solapan con las cadenas en virtud de:

```
char test_array[] = "HELLO!";
char *string_ptr  = "HELLO!";
```

El caso 1 crea una secuencia de bytes con el equivalente ASCII de "HELLO!". De igual forma, el segundo caso ubica la misma secuencia de bytes, pero además crea un puntero llamado string_ptr que apunta a la misma secuencia de caracteres.

El caso 2 equivale realmente a:

```
char test_array[] = "HELLO!";
```

Y en tiempo de ejecución:

```
char string_ptr = test_array;  // Nombre de array tratado como puntero
o;

char string_ptr = &test_array[0]; // Pone la dirección del primer
                                  // elemento del array en el puntero
```

Esto muestra nuevamente la equivalencia entre punteros y arrays. La primera sentencia significa "poner la dirección de test_array en string_ptr". En este contexto el nombre del array se trata como un apuntador a su primer elemento, haciendo innecesario el empleo del operador de dirección (&).

La segunda sentencia significa "obtener la dirección del primer elemento del array y ponerla en string_ptr". No hay ninguna conversión implícita a puntero, sólo la obtención de la dirección base del array.

6.4.3.- Uso de Arrays

```
    /* Copiar la cadena HELLO! En un array vacío */

    unsigned char source_array[] = "HELLO!";
    unsigned char dest_array[7];
    unsigned char array_index ;

    array_index = 0 ;

    while(array_index < 7) {  // Comprobar el final del array
       dest_array[array_index] = source_array[array_index] ;
       //Mover carácter-a-carácter al array destino
        array_index++ ;
       }
```

La variable array_index contiene el desplazamiento, medido desde el comienzo del array, del carácter a mover, y del carácter a guardar.

Como ha sido indicado, los punteros y arrays se encuentran íntimamente relacionados. El programa anterior puede re-escribirse así:

```
/* Copiar la cadena HELLO! En un array vacío */

char *string_ptr = "HELLO!" ;
unsigned char dest_array[7] ;
unsigned char array_index  ;
array_index = 0 ;

while(array_index < 7) {      // Comprobar el final del array
   dest_array[array_index] = string_ptr[array_index] ;
```

```
//Mover carácter-a-carácter al array destino
array_index++ ;
}
```

Notar que quitando el '*' a string_ptr y añadiendo el par '[ ]', el puntero se convierte en un array! Sin embargo en este caso se puede copiar la cadena HELLO!, utilizando la forma *ptr++:

```
array_index = 0 ;

while(array_index < 7) { // Comprobar el final del array
   dest_array[array_index] = *string_ptr++ ;
   //Mover carácter-a-carácter al array destino
   }
```

La sentencia "c2 = *ptr++", no significa "incrementar el valor apuntado por ptr, y asignarlo a c2", sino "asignar a c2 el valor apuntado por ptr, e incrementar ptr para que apunte al siguiente char".

6.4.4.- Resumen sobre Arrays y Punteros

Para resumir

Crear un puntero a char que todavía no apunta a ningún lugar:

```
char *x_ptr ;
```

Crear un puntero a una variable C:

```
char x ;
char *x_ptr = &x ;
```

Crear un array no inicializado:

```
unsigned char x_arr[10] ;
```

Crear e inicializar un array:

```
unsigned char x_arr[] = { 0,1,2,3 } ;
```

Crear un array de caracteres o cadena:

```
char x_arr[] = "HELLO" ;
```

Crear un puntero a cadena:

```
char *string_ptr =  "HELLO" };
```

Crear un puntero a un array:

```
char x_arr[] = "HELLO" ;
char *x_ptr = x_arr ;
```

6.5.- ESTRUCTURAS

Las estructuras es quizás lo que hace de C un potente lenguaje que permite crear complejos programas capaces de manejar grandes cantidades de datos. Se trata básicamente de una forma de agrupar datos relacionados bajo un nombre simbólico.

6.5.1.- ¿Por qué usar estructuras?

Veamos un ejemplo: Un programa en C51 debe realizar la linealización de una señal procedente de una variedad de sensores de presión fabricados por la misma compañía. Cada sensor lleva asociada una señal de entrada con su *span* y *offset*, un coeficiente de temperatura, y un amplificador acondicionador de señal de una determinada ganancia. La información de cada sensor puede almacenarse en variables normales así:

```
unsigned char sensor_type1_gain = 0x30 ;
unsigned char sensor_type1_offset = 0x50 ;
unsigned char sensor_type1_temp_coeff = 0x60 ;
unsigned char sensor_type1_span = 0xC4 ;
unsigned char sensor_type1_amp_gain = 0x21 ;

unsigned char sensor_type2_gain = 0x32 ;
unsigned char sensor_type2_offset = 0x56 ;
unsigned char sensor_type2_temp_coeff = 0x56 ;
unsigned char sensor_type2_span = 0xC5 ;
unsigned char sensor_type2_amp_gain = 0x28 ;

unsigned char sensor_type3_gain = 0x20 ;
unsigned char sensor_type3_offset = 0x43 ;
unsigned char sensor_type3_temp_coeff = 0x61 ;
unsigned char sensor_type3_span = 0x89 ;
unsigned char sensor_type3_amp_gain = 0x29 ;
```

Como se puede ver, todos los nombres siguen el patrón:

```
unsigned char sensor_typeN_gain = 0x20 ;
unsigned char sensor_typeN_offset = 0x43 ;
unsigned char sensor_typeN_temp_coeff = 0x61 ;
unsigned char sensor_typeN_span = 0x89 ;
unsigned char sensor_typeN_amp_gain = 0x29 ;
```

Donde 'N' es el número del tipo de sensor. Una estructura es la manera ordenada de condensar este tipo de datos. De echo, la información necesaria para describir a cada sensor se reduce a:

```
unsigned char gain ;
unsigned char offset ;
unsigned char temp_coeff ;
unsigned char span ;
unsigned char amp_gain ;
```

El concepto de estructura reside en la idea de utilizar una plantilla general para datos relacionados. Para el ejemplo anterior se declara la estructura sensor_desc:

```
struct sensor_desc {unsigned char gain ;
                    unsigned char offset ;
                    unsigned char temp_coeff ;
                    unsigned char span ;
```

```
                unsigned char amp_gain ; } ;
```

La declaración de la estructura anterior no realiza ninguna asignación de memoria. Simplemente crea una plantilla que puede utilizarse para poner datos en memoria, como se muestra a continuación:

```
struct sensor_desc sensor_database ;
```

Lo cual significa "utilizar la plantilla sensor_desc para reservar un área de memoria para el elemento sensor_database, que se refiere a un conjunto de valores similares a los de la plantilla". De esta forma se crea una estructura formada por un grupo de 5 unsigned char.

El acceso a los elementos individuales de la estructura se realiza así:

```
sensor_database.gain = 0x30 ;
sensor_database.offset = 0x50 ;
sensor_database.temp_coeff = 0x60 ;
sensor_database.span = 0xC4 ;
sensor_database.amp_gain = 0x21 ;
```

6.5.2.- Arrays de estructuras

Siguiendo con el ejemplo anterior, la información de varios tipos de sensores se puede agrupar en un array, de igual forma que se hacía con los int y char:

```
struct sensor_desc sensor_database[4] ;
```

Esta sentencia crea cuatro estructuras en memoria, cada una de las cuales es semejante a la plantilla creada. El acceso a los elementos del array requiere ahora un índice:

```
/* Para trabajar con el Sensor 0 (Primer elemento de la estructura */

sensor_database[0].gain = 0x30 ;
sensor_database[0].offset = 0x50 ;
sensor_database[0].temp_coeff = 0x60 ;
sensor_database[0].span = 0xC4 ;
sensor_database[0].amp_gain = 0x21 ;

/* Para trabajar con el Sensor 1 (Segundo elemento de la estructura */

sensor_database[1].gain = 0x32 ;
sensor_database[1].offset = 0x56 ;
sensor_database[1].temp_coeff = 0x56 ;
sensor_database[1].span = 0xC5 ;
sensor_database[1].amp_gain = 0x28 ;

y así sucesivamente...
```

6.5.3.- Inicialización de estructuras

Al igual que sucede con los arrays, las estructuras pueden inicializarse en su declaración:

```
struct sensor_desc sensor_database = { 0x30, 0x50, 0x60, 0xC4, 0x21 }
;
```

Y el caso de un array de estructuras::

```
struct sensor_desc sensor_database[4] =
 { {0x30,0x50,0x60,0xC4,0x21},
    ...
   {0x32,0x56,0x56,0xC5,0x28}
 } ;
```

6.5.4.- Ubicación de estructuras en direcciones absolutas

Algunas veces es necesario colocar una estructura en una dirección absoluta. Imaginemos un chip de reloj mapeado en memoria. La estructura RTC sería:

```
Contenido del fichero RTCBYTES.C

struct RTC { unsigned char segundos;
             unsigned char minutos ;
             unsigned char horas;
             unsigned char dias;
} ;

struct RTC xdata RTC_chip ;  // Crea una estructura en xdata
```

Aquí se utiliza un fichero con la idea de usar el linker para asignar una dirección absoluta en XRAM a la estructura RTC_chip, tal como se muestra seguidamente:

```
/* Estructura ubicada en la dirección del chip de reloj */

Fichero MAIN.C

extern xdata struct RTC_chip ;

/* Otros objetos XDATA */

xdata unsigned char time_segs, time_mins ;

void main(void) {

time_segs = RTC_chip.segundos;
time_mins = RTC_chip.minutos;
}
```

La orden al linker para ubicar la estructura RTC_chip en la dirección del chip de reloj es:

```
l51 main.obj,rtcbytes.obj XDATA(?XD?RTCBYTES(8000h))
```

Ver sección 7.6 para otros ejemplos de este método de ubicación.

6.5.5.-.Punteros a estructuras

Veamos un ejemplo de acceso a estructuras mediante punteros:

```
/* Definición de puntero a estructura */

struct sensor_desc *sensor_database ;

/* Uso del puntero para acceder a los elementos de la estructura */

sensor_database->gain = 0x30 ;
```

```
sensor_database->offset = 0x50 ;
sensor_database->temp_coeff = 0x60 ;
sensor_database->span = 0xC4 ;
sensor_database->amp_gain = 0x21 ;
```

Es importante destacar el uso del operador '->' para acceder a los elementos de una estructura mediante un puntero.

6.5.6.- Paso a funciones de punteros a estructuras

Los punteros a estructuras se utilizan frecuentemente en las llamadas a funciones, para evitar el paso de la estructura completa. De esta forma, antes de llamar a la función, solo se necesita empilar los 2 ó 3 bytes del puntero, en lugar de los 20 bytes o más que puede ocupar una estructura. El procedimiento sería:

```
struct sensor_desc *sensor_database ;

sensor_database-> gain = 0x30 ;
sensor_database-> offset = 0x50  ;
sensor_database-> temp_coeff = 0x60 ;
sensor_database-> span = 0xC4 ;
sensor_ database- >amp_gain = 0x21 ;

test_function(*struct_pointer) ;

test_function(struct sensor_desc *received_struct_pointer) {
   received_struct_pointer->gain = 0x20 ;
   received_struct_pointer->temp_coef = 0x40 ;
   }
```

Nota: Cuando se utiliza un puntero en una llamada a función, se permite que la función opere directamente sobre la estructura y no sobre una copia de la misma.

6.5.7.- Punteros a estructuras en direcciones absolutas

Algunas veces es necesario colocar una estructura en una dirección absoluta. Imaginemos un chip de reloj mapeado en memoria. Un método alternativo al estudiado en la sección 6.4.4. es acceder al chip de reloj mediante un puntero a estructura.

Una diferencia importante es que en este caso no se reserva memoria para la estructura. La plantilla en este caso sería:

```
/* Definición de la estructura RTC */

struct RTC {char segundos;
        char minutos;
        char horas;
        char dias; } ;

/* Creación de un puntero a estructura */

struct RTC xdata *rtc_ptr ;  // 'xdata' indica a C51 que se trata
                     // de un dispositivo mapeado en memoria.


void main(void) {
```

```
   rtc_ptr = (void xdata *) 0x8000 ;  // rtc_ptr apunta
                           // a la dirección 0x8000 de xdata
                           // donde está el chip de reloj
    rtc_ptr->segundos = 0 ;  // Opera con los elementos
    rtc_ptr->minutos = 0x01 ;
   }
```

Se trata de una técnica general que puede utilizarse en cualquier situación que precise de un puntero sobre una estructura residente en un dispositivo de IO. Sin embargo, es responsabilidad del usuario el asegurarse de que el linker no utilice esas direcciones para ubicar variables en RAM.

Resumiendo, el procedimiento es:

1. Definir una plantilla
2. Declarar un puntero a estructura
3. En tiempo de ejecución, asignar al puntero una dirección absoluta.

6.6.- UNIONES

Una unión es un concepto similar al de una estructura salvo que en lugar de utilizar posiciones consecutivas de memoria para los distintos elementos de la estructura, los elementos de la unión ocupan las mismas direcciones de memoria. Así una unión de 4 bytes sólo ocupa un byte. Una unión puede estar formada por un long, un char y un int. En este caso el número de bytes de RAM utilizados por la unión viene determinado por el tamaño del elemento más grande (el long). En el siguiente ejemplo:

```
union test { char x ;
             int y  ;
             char a[3] ;
             long z ;
} ;
```

la unión requiere 4 bytes. La ubicación de cada elemento es:

```
dirección   0   x byte  y byte alto  a[0]  z byte más alto
          +1          y byte bajo  a[1]  z byte
          +2                       a[2]  z byte
          +3                             z byte más bajo
```

Recordemos que el 8051 coloca el byte más significativo de un objeto en la dirección de memoria más baja.

En las aplicaciones con el 8051, las uniones se utilizan frecuentemente para acceder a los diferentes bytes de un int o un long, como en el siguiente ejemplo:

```
/* Declaración de la unión clock */

union clock {long real_time_count ; // Reservar 4 bytes
    int real_time_words[2] ;        // array de 2 int
    char real_time_bytes[4] ;       // array de 4 char
      } ;

/* Real Time Interrupt */

void timer0_int(void) interrupt 1 using 1 {
```

```
clock.real_time_count++ ;       // Incrementar clock

if(clock.real_time_words[1] == 0x8000) { // Comprobar
                // si el word bajo es 0x8000

/* Hacer algo! */
}

if(clock.real_time_bytes[3] == 0x80) {  // Comprobar
               // si el byte bajo es 0x80

/* Hacer algo! */
}

}
```

6.7.- PUNTEROS GENÉRICOS

El compilador C51 permite operar con punteros genéricos, y desde la versión 3.00 también con punteros específicos.

Un puntero genérico puede apuntar a cualquier objeto y en cualquier área de memoria. Ocupa 3 bytes: el primero para indicar el tipo de memoria al que apunta (1=idata, 2=xdata, 3=pdata, 4=data, 5=code), y los otros dos para almacenar la dirección del objeto apuntado.

Un puntero específico apunta a objetos residentes en tipos concretos de memoria. Para su almacenamiento requiere: 1 byte (punteros a data, idata, y pdata), o 2 bytes (punteros a xdata y code). Los punteros específicos son muy eficientes y reducen el tamaño de código empleado por el compilador.

Ejemplo, al compilar el siguiente código:

```
xdata char buffer[10] ;
code char message[] = "HELLO" ;
void main(void) {
      char *s ;
      char *d ;

      s = message ;
      d = buffer ;

      while(*s != '\0') {
         *d++ = *s++ ;
         }
  }
```

Se obtiene:

```
    RSEG  ?XD?T1
buffer:           DS  10
    RSEG  ?CO?T1
message:
    DB  'H' ,'E' ,'L' ,'L' ,'O' ,000H
;
;
```

```
; xdata char buffer[10] ;
; code char message[] = "HELLO" ;
;
;    void main(void) {
    RSEG  ?PR?main?T1
    USING    0
main:
;
;       char *s ;
;       char *d ;
;
;       s = message ;
    MOV      s?02,#05H
    MOV      s?02+01H,#HIGH message
    MOV      s?02+02H,#LOW message
;       d = buffer ;
    MOV      d?02,#02H
    MOV      d?02+01H,#HIGH buffer
    MOV      d?02+02H,#LOW buffer
?C0001:
;
;       while(*s != '\0') {
    MOV      R3,s?02
    MOV      R2,s?02+01H
    MOV      R1,s?02+02H
    LCALL    ?C_CLDPTR
    JZ       ?C0003
;          *d++ = *s++ ;
    INC      s?02+02H
    MOV      A,s?02+02H
    JNZ      ?C0004
    INC      s?02+01H


?C0004:
    DEC      A
    MOV      R1,A
    LCALL    ?C_CLDPTR
    MOV      R7,A
    MOV      R3,d?02
    INC      d?02+02H
    MOV      A,d?02+02H
    MOV      R2,d?02+01H
    JNZ      ?C0005
    INC      d?02+01H
?C0005:
    DEC      A
    MOV      R1,A
    MOV      A,R7
    LCALL    ?C_CSTPTR
;         }
    SJMP     ?C0001
;       }
?C0003:
    RET
; END OF main
    END
```

Como se puede apreciar, los punteros '*s' y '*d' ocupan 3 bytes. El byte '05' de s indica que se trata de un puntero a un objeto en code, mientras que el byte '02' señala que el puntero d apunta a un objeto en xdata. Los bytes que indican el tipo de memoria apuntado son:

```
CODE  - 05
XDATA - 02
PDATA - 03
DATA  - 04
IDATA - 01
```

6.8.- PUNTEROS ESPECÍFICOS EN C51

Como ya se ha comentado, los punteros específicos apuntan a objetos residentes en tipos concretos de memoria. Para su almacenamiento requieren: 1 byte (punteros a idata, data y pdata), o 2 bytes (punteros a xdata y code). Los punteros específicos son muy eficientes y reducen el tamaño de código empleado por el compilador. Un puntero específico a char residente en xdata se define así:

```
char xdata *ext_ptr ;
```

y un puntero a char en code, de esta forma:

```
char code *const_ptr ;
```

En ambos casos, los punteros se almacenan en el área de memoria correspondiente al modelo de memoria utilizado (SMALL en data, COMPACT en pdata, LARGE en xdata). Si se desea crear un puntero a char en xdata, pero que el propio puntero resida en pdata, la declaración debe ser:

```
char xdata * pdata ext_ptr ;
```

En el siguiente ejemplo, al igual que en el ejemplo anterior, se copia una cadena desde CODE hacia XDATA. En este caso el tiempo que se tarda en copiar la cadena se reduce un 50% con respecto al caso anterior. La nueva strcpy() se ha llamado strcpy_x_c().

El prototipo de la función es:

extern char xdata *strcpy_x_c(char xdata*,char code *) ;

y a continuación se ofrece el código generado:

```
; char xdata *strcpy_x_c(char xdata *s1, char code *s2)  {
_strcpy_x_c:
    MOV      s2?10,R4
    MOV      s2?10+01H,R5
;__ Variable 's1?10' assigned to Register 'R6/R7' __
;   unsigned char i = 0;
;__ Variable 'i?11' assigned to Register 'R1' __
    CLR      A
    MOV      R1,A
?C0004:
;
;   while ((s1[i++] = *s2++) != 0);
    INC      s2?10+01H
    MOV      A,s2?10+01H
    MOV      R4,s2?10
    JNZ      ?C0008
```

```
    INC       s2?10
?C0008:
    DEC       A
    MOV       DPL,A
    MOV       DPH,R4
    CLR       A
    MOVC      A,@A+DPTR
    MOV       R5,A
    MOV       R4,AR1
    INC       R1
    MOV       A,R7
    ADD       A,R4
    MOV       DPL,A
    CLR       A
    ADDC      A,R6
    MOV       DPH,A
    MOV       A,R5
    MOVX      @DPTR,A
    JNZ       ?C0004
?C0005:
;   return (s1);
; }
?C0006:
    END
```

Capítulo 7 - Acceso a los dispositivos externos mapeados en memoria

7.1.- INTRODUCCIÓN

Con frecuencia se añaden puertos de IO al 8051 para compensar la pérdida de los puertos P0 y P2 que se produce al aumentar la memoria del µC. Habitualmente se dispone el hardware de forma que los puertos adicionales aparezcan como bytes de RAM externa, que pueden leerse y escribirse con instrucciones MOVX. Se dice que dispositivos externos se encuentran mapeados en memoria, cuando se accede a los mismos en la forma descrita. Es muy común mapear en memoria *xdata* del 8051 dispositivos tales como: UARTS, puertos de IO, o chips de reloj en tiempo real.

La forma más sencilla de conseguirlo es uniendo las líneas /RD y/o /WR al dispositivo. Suponiendo que solo haya un dispositivo, no es necesario recurrir a un decodificador de direcciones. Para acceder al dispositivo desde C, basta con declarar la variable de acceso al dispositivo con la palabra *xdata*. Así el compilador utilizará las instrucciones MOVX A,@DPTR y MOVX @DPTR,A para leer y escribir en él. El linker tratará de asignar una ubicación a la variable, pero como no existe ningún decodificador, el dispositivo será habilitado con las señales /RD o /WR.

En la realidad este caso tan sencillo se presenta rara vez. Lo típico es encontrarse con una mezcla de RAM, UARTs, puertos, EEPROM y otros dispositivos, mapeados en el espacio *xdata* del 8051. En estos casos se utiliza lógica (decodificador o una PAL) para regular el acceso a los periféricos.

En esta situación, cada dispositivo aparece en unas posiciones fijas del espacio *xdata*. Lo ideal sería poder referirse a estos dispositivos por su nombre, al igual que se hace con los periféricos internos del 8051. Hay tres formas diferentes de lograrlo:

1. Usar variables normales, char, int, etc, ubicadas apropiadamente por el linker
2. Usar apuntadores y desplazamientos, con macros XBYTE, o directamente con apuntadores definidos por el usuario.
3. Usar las directiva _at_ y _ORDER.

Las siguientes secciones tratan detalladamente estas alternativas.

7.2.- LAS MACROS XBYTE Y XWORD

Para permitir que los dispositivos mapeados en memoria, sean accesibles desde C, se requiere un método para que los apuntadores señalen a direcciones fijas de memoria. Con C51 hay varias formas de lograrlo, la más sencilla de las cuales consiste en utilizar las macros XBYTE[dir16] y XWORD[dir16].

Por ejemplo, el acceso al registro PORT8_DDI de un dispositivo de IO mapeado en la dirección 0x8000 de memoria, se realiza de la siguiente manera:

```
    #include "absacc.h";  /* Contiene las definiciones de las macros
*/
    #define port8_ddi   XBYTE[0x8000]
```

```
    #define port8_data  XBYTE[0x8001]
```

Y se usan así:

```
    port8_ddi = 0xFF       ;
    input_val = port8_data ;
```

Para acceder a una palabra situada en una dirección par de memoria externa:

```
    #define word_reg XWORD[0x4000] /* Palabra en dirección 0x8000 */
/* pues la dirección byte 0x8000 corresponde a la dirección word
0x4000 */
```

La dirección indicada por "word_reg" es fija y queda fijada en tiempo de compilación. Dentro de los corchetes solo se acepta una constante.

No es posible cambiar la dirección apuntada con métodos basados en el uso de estas macros. Sin embargo, es el mejor método para direccionar posiciones fijas en el hardware, que no van a cambiar en tiempo de ejecución.

Prescindiendo de la macro XWORD, la línea C equivalente sería:

```
    #define word_reg_ptr ((unsigned int *) 0x24000L)
/* crea un puntero genérico a unsigned en xdata apuntando a dir 0x8000
*/
/* pues la dirección byte 0x8000 corresponde a la dirección word
0x4000 */
```

Que se utilizaría así:

```
    *word_reg_ptr = 0xFFFF ;
```

Veamos el código generado en algunos casos:

```
; #define XBYTE ((unsigned char volatile *) 0x20000L)
; #define XWORD ((unsigned int volatile *) 0x20000L)
;
; main() {

        RSEG  ?PR?main?T2
        USING   0
main:
; char x ;
; int  y ;
;
; x = XBYTE[0x8000]       ;
        MOV     DPTR,#08000H
        MOVX    A,@DPTR
        MOV     x?040,A
;
; y = XWORD[0x8000/sizeof(int)] ;
        MOVX    A,@DPTR
        MOV     R6,A
        INC     DPTR
        MOVX    A,@DPTR
        MOV     y?041,R6
        MOV     y?041+01H,A
```

```
; }
        RET
; END OF main
```

El uso del tipo de almacenamiento **volatile**, resulta esencial para evitar que el optimizador elimine lecturas de datos desde los puertos. Véase la Capítulo 7 "El tipo de almacenamiento "volatile" " para más detalles.

Nota: es necesario incluir el fichero de encabezamiento "absacc.h", que contiene la definición de las macros XBYTE, XWORD, etc.

7.3.- INICIALIZACIÓN DE PUNTEROS A XDATA EN TIEMPO DE COMPILACIÓN

En ocasiones se necesita un apuntador que en tiempo de compilación apunte a una dirección concreta, pero que permita, más adelante, apuntar a una dirección diferente. En estos casos no puede hacerse uso de las macros anteriores y se necesita recurrir a los punteros. La inicialización de los mismos se haría del siguiente modo:

```
char xdata *a_ptr = 0x8000; /* a_ptr es un puntero específico a char
en xdata */

char *a_ptr = 0x028000L;  /* a_ptr es ahora un puntero genérico a char
*/

char * xdata a_ptr = 0x028000L;  /* puntero genérico a char, residente
en xdata */
```

En los dos ejemplos anteriores el puntero señala inicialmente a la dirección 0x8000 de *xdata*. En el primer ejemplo, el puntero sólo puede apuntar a posiciones *xdata*. El puntero del segundo ejemplo puede apuntar a cualquier espacio de memoria. El valor "02", en el caso del puntero genérico le informa a C51 que debe apuntar a dirección *xdata*.

Los punteros son variables que contienen la dirección de otra variable. En los dos primeros ejemplos, los punteros mismos residen en el área correspondiente al modelo de memoria utilizado (*data* para el modelo SMALL). En el tercer ejemplo, el propio puntero reside en *xdata*, puesto que la palabra *xdata* se encuentra a la derecha del asterisco.

En el siguiente ejemplo ptr es un puntero a char en memoria *xdata*, y el propio puntero reside en *xdata*:

```
; char xdata * xdata ptr = 0x8000 ;
;
; main()

main:
;   {
;   char x ;

;   ptr += 0xf0 ;
        MOV     DPTR,#ptr+01H
        MOVX    A,@DPTR
        ADD     A,#0F0H
```

```
        MOVX    @DPTR,A
        MOV     DPTR,#ptr
        MOVX    A,@DPTR
        ADDC    A,#00H
        MOVX    @DPTR,A

;   x = *ptr ;
        MOVX    A,@DPTR
        MOV     R6,A
        INC     DPTR
        MOVX    A,@DPTR
        MOV     DPL,A
        MOV     DPH,R6
        MOVX    A,@DPTR
        MOV     x?040,A
;   }
        RET
; END OF main
```

7.4.- INICIALIZACIÓN DE PUNTEROS A XDATA EN TIEMPO DE EJECUCIÓN

Con frecuencia se necesita apuntar a direcciones en *xdata*, que solo son conocidas en tiempo de ejecución. En consecuencia, la asignación de valores a estos punteros debe hacerse cuando el programa está corriendo. Inicialmente, en tiempo de compilación, se define el puntero y no se le asigna ningún valor inicial. La asignación se realiza posteriormente, en tiempo de ejecución:

```
char xdata *cx_ptr ;   /* Puntero a char en xdata sin inicializar */

main() {

cx_ptr = (char xdata*)0x8000 ; /*Apunta a dirección 0x8000 de xdata */
}
```

Otra posibilidad consiste en declarar un puntero a *xdata* y asignarle el valor de una variable. Veamos un ejemplo:

```
; char xdata *ptr ; /* Puntero a char en xdata */
;
; void  main(void) {

main:
;    char c1;
;    int dir_inicio = 0x8000;  /* Variable que contiene una
dirección */
;---- Variable 'dir_inicio?041' assigned to Register 'R6/R7' --
--
       MOV     R7,#00H
       MOV     R6,#080H
;    ptr = (char *)dir_inicio ;
       MOV     ptr,R6
       MOV     ptr+01H,R7
?C0001:
;    while(1)
```

```
;       c1 = *ptr++ ;
        INC     ptr+01H
        MOV     A,ptr+01H
        MOV     R6,ptr
        JNZ     ?C0004
        INC     ptr
?C0004:
        DEC     A
        MOV     DPL,A
        MOV     DPH,R6
        MOVX    A,@DPTR
        MOV     c1?040,A
        SJMP    ?C0001
;    }
        RET
; END OF main
```

Una variación consiste en declarar un puntero a 0x0000 y utilizar una variable como índice:

```
; char xdata *ptr ; /* Puntero a char en xdata */
;
;
; void main(void) {

main:
;    unsigned char uc1, uc2;
;    ptr = (char *) 0x0000 ;
       CLR     A
       MOV     ptr,A
       MOV     ptr+01H,A
;    for(uc1 = 0; uc1 < 0x40 ; uc1++)
;---- Variable 'uc1?040' assigned to Register 'R7' ----
       MOV     R7,A
?C0001:
;       uc2 = ptr[uc1] ;
       MOV     A,ptr+01H
       ADD     A,R7
       MOV     DPL,A
       CLR     A
       ADDC    A,ptr
       MOV     DPH,A
       MOVX    A,@DPTR
       MOV     uc2?041,A
       INC     R7
       CJNE    R7,#040H,?C0001
;    }
?C0004:
       RET
; END OF main
```

7.5.- EL TIPO DE ALMACENAMIENTO "VOLATILE"

Una situación muy corriente con dispositivos externos es que los valores de sus registros cambian sin intervención de la CPU. Un buen ejemplo es el chip de reloj, los registros que almacenan el tiempo transcurrido cambian continuamente sin que intervenga la CPU. Considérese lo siguiente:

```
unsigned int xdata *milisegundos = 0x8000 ; // Puntero al chip de
reloj

tiempo = *milisegundos ; -> (1)  // Leer valor del registro del reloj

array[x] = tiempo ;

tiempo = *milisegundos ;  -> (2)  // Segundo acceso al registro

array[y] = tiempo ;
```

Aquí el valor escrito en el array es el valor de *milisegundos, un registro del chip de reloj.

El programa se compila sin errores pero no trabaja bien ¿Por qué?. La razón reside en que el optimizador de compilador da por supuesto que al no haberse producido ninguna escritura entre (1) y (2), el valor de *milisegundos no ha cambiado. Por ello optimiza el código correspondiente al segundo acceso haciendo que array[y] == array[x].

La solución es declarar *milisegundos como "volatile", así:

unsigned int volatile xdata *milisegundos = 0x8000 ;

De esta forma el optimizador no eliminará el código correspondiente al segundo acceso.

7.6.- UBICACIÓN DE VARIABLES EN DIRECCIONES ESPECÍFICAS

7.6.1.- Utilizando el Linker

Un método para ubicar variables externas, especialmente arrays, en direcciones fijas consiste en utilizar el linker en lugar del compilador. Por ejemplo, para producir un array de 30 caracteres a partir de la dirección 0x8000 de RAM externa, se deben seguir los siguientes pasos:

```
/*** Módulo 1 ***/

/* Este módulo contiene solo declaraciones de variables */

unsigned char xdata array[30] ;

/* Fin del Módulo 1 */

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

/*** Módulo 2 ***/

/* Este módulo contiene sentencias ejecutables */

extern unsigned char xdata array[30] ;
```

```
main()

{
unsigned char i ;

i = array[i] ;

}
```

Ahora, llamando al linker de la siguiente forma:

```
L51 modulo1.obj, modulo2.obj XDATA (?XD?modulo1 (8000H))
```

el linker hará que el segmento XDATA del Módulo 1 (indicado por ?XD?modulo1) comience en la dirección 0x8000, independientemente de cualquier otra declaración de variables xdata. De esta forma, el array comienza en dirección 0x8000 y tiene 30 bytes de longitud.

Se puede ejercer un control similar sobre las direcciones de los segmentos en cualquier otro espacio de memoria, sabiendo que C51 utiliza los siguientes nombres para los nombres de segmento:

```
CODE   ?PR?NombreDeFuncion?NombreDeFichero  (código ejecutable)
CODE   ?CO?NombreDeFuncion?NombreDeFichero  (tablas, etc.)
BIT    ?BI?NombreDeFuncion?NombreDeFichero
DATA   ?DT?NombreDeFuncion?NombreDeFichero
XDATA  ?XD?NombreDeFuncion?NombreDeFichero
PDATA  ?PD?NombreDeFuncion?NombreDeFichero
```

Así el área de parámetros (modelo LARGE) de la función 'test()' del fichero MOD1.C será:

```
?XD?TEST?MOD1,
```

y el código:

```
?PR?TEST?MOD1
```

El conocimiento de estos datos es útil para interrelacionar programas en ensamblador y en lenguaje C. Véase el capítulo 14.

7.7.- EXCLUSIÓN DE RANGOS DE MEMORIA EN XDATA

En gran parte lo que sigue tiene relación con la sección anterior. Ocasionalmente un dispositivo mapeado en memoria, tal como un chip de reloj, se utiliza como reloj y como RAM. Típicamente los primeros 8 bytes del chip están relacionados con el reloj, y se utilizan para almacenar los segundos, minutos, etc, y los restantes 248 bytes son RAM.

Si al linker L51 no se le informa en sentido contrario, coloca las variables *xdata* a partir de la dirección cero. Y si el reloj está mapeado en dirección 0, los accesos a las variables *xdata* sobre-escribirán las posiciones destinadas al reloj. Además suele ser conveniente referirse a los registros de forma individual.

Una solución consiste en definir un módulo especial que contenga solamente una estructura que describa los registros del reloj. En el programa principal, los accesos a los registros del reloj se hacen igual que a los elementos de una estructura. El truco para evitar la sobre-escritura descrita antes, consiste en asignar a este módulo especial la dirección 0, durante el proceso de linkado. Así el linker coloca esta estructura en la dirección 0, y a continuación coloca el resto de las variables XDATA. Este método puede utilizarse para evitar que L51 coloque variables en determinadas áreas de memoria. Veamos un ejemplo de exclusión de áreas específicas con L51:

```
/* Estructura situada en la dirección de base del chip de reloj */

Módulo MAIN.C

extern xdata struct {   unsigned char segundos;
                unsigned char minutos;
                unsigned char horas;
                unsigned char dias; } RTC_chip ;

/* Otros objetos XDATA */

xdata unsigned char uc_seg, uc_min;

void main(void) {

uc_seg = RTC_chip.segundos ;
uc_min = RTC_chip.minutos ;

}
```

Módulo RTCBYTES.C

```
xdata struct { unsigned char segundos;
               unsigned char minutos ;
               unsigned char horas   ;
               unsigned char dias    ; } RTC_chip ;
```

Llamada al Linker para ubicar la estructura RTC_chip sobre los registros reales del reloj:

```
l51 main.obj,rtcbytes.obj XDATA(?XD?RTCBYTES(0h))
```

7.8.- LOS OLVIDADOS ORDER Y _at_ AHORA EN C51

Quizás una de las omisiones más curiosas de C51 fue su incapacidad (hasta la versión v4.xx) para asignar a un objeto una dirección absoluta desde el fichero fuente. Aunque existen otros procedimientos para lograrlo, los usuarios han demandado largamente una extensión al lenguaje C, que permitiera asignar una dirección absoluta a un objeto desde el código fuente. Afortunadamente el control _at_ ya existe.

7.9.- USO DE LOS CONTROLES _at_ Y _ORDER_

Aquí. el orden de las variables no debe cambiar y se deben corresponder con las posiciones de los registros del reloj. La directiva #pragma ORDER ordena a C51 que coloque los objetos en orden de direcciones crecientes, con el primer elemento en la dirección más baja. El linker se utiliza para fijar la dirección de todo el bloque en memoria.

Módulo MAIN.C

```
#pragma ORDER
unsigned char xdata RTC_segundos ;
unsigned char xdata RTC_minutos ;
unsigned char xdata RTC_horas ;

main() {   RTC_minutos = 1 ; }
```

Fichero de entrada para el linker MAIN.LIN

```
main.obj  to main  XDATA(?XD?MAIN(0fa00h))
```

Alternativamente, el control _at_ obliga a que C51 ponga los objetos en las direcciones especificadas en el fichero fuente:

```
unsigned char xdata RTC_segundos _at_ 0xfa00 ;
unsigned char xdata RTC_minutos _at_ 0xfa01 ;
unsigned char xdata RTC_horas _at_ 0xfa02 ;

main()   {    RTC_minutos = 1 ;
      }
```

...lo que felizmente resulta auto-explicativo.

Capítulo 8 - Tareas del linker y ubicación de la pila

8.1.- USO BÁSICO DEL LINKER L51

Estos temas originan cierto grado de confusión, sobre todo en los usuarios de otros compiladores.

El linker, L51 en el caso de Keil o RL51 en el caso de Intel, es el encargado de combinar los distintos módulos de un programa C. El compilador no asigna direcciones definitivas a las líneas de código que produce, tan solo genera un desplazamiento desde el comienzo del fichero. Obviamente, el código antes de ser ejecutado debe tener asignada una dirección en memoria de código. Este trabajo de asignación también lo realiza el linker que además busca en las librerías el código necesario y lo inserta en el programa ejecutable.

Un ejemplo de invocación del linker es:

l51 startup.obj, modulo1.obj, modulo2.obj, modulo3.obj, C51L.lib to exec.abs

Aquí, el linker combina tres módulos de código objeto y el fichero startup.obj, buscando las funciones de librería necesarias en C51L.lib. Para lograrlo, deberá ajustar las direcciones definitivas de todas las instrucciones JMP y CALL. Como resultado de la combinación de los cinco ficheros anteriores, produce un fichero absoluto o ejecutable llamado EXEC.ABS. El linker también genera otro fichero llamado EXEC.M51, que contiene un resumen de las operaciones realizadas por el linker, y lo que es más importante, con la dirección de todos los símbolos utilizados en el programa, y el tamaño de cada módulo.

Salvo en las aplicaciones sencillas, el número de parámetros que necesita el linker en su invocación puede ser muy elevado. En el sistema operativo DOS, la longitud máxima de la línea de llamada es de 128 caracteres. Si esta longitud resulta insuficiente, es posible utilizar varias líneas terminando cada línea con un & seguido de un retorno de carro. Para facilitar el proceso en las sucesivas llamadas que habitualmente se producen durante la depuración, es posible especificar los parámetros en un fichero ASCII, con lo que la llamada al linker se reduce a:

```
   l51 @fichero_de_entrada
```

Siendo el contenido del fichero_de_entrada el siguiente:

```
   startup.obj,&
   module1.obj,&
   module2.obj,&
   module3.obj,&
   &
   C51L.lib &
   &
   to exec.abs
```

El linker dispone de controles para determinar la dirección en la cual debe ubicar cada tipo de memoria. Por ejemplo, si la RAM externa del chip comienza en dirección 0x4000, y la memoria de código (EPROM) en la 0x8000, la llamada al linker debe ser:

```
l51 startup.obj, module1.obj, module2.obj, module3.obj,
C51L.lib to exec.abs CODE(8000H) XDATA(4000H)
```

Con ello, todas las variables de RAM externa se colocarán a partir de la dirección 0x4000, y el programa ejecutable a partir de la dirección 0x8000.

Nota: El nuevo entorno de desarrollo para Windows, desarrollado por Keil, y denominado µVision, incluye un editor, un gestor de proyectos, y un gestor de aplicaciones. El nuevo entorno de trabajo, ayuda a mantener actualizados los proyectos cada vez que se realizan cambios en los ficheros fuente, y elimina la necesidad de aprender la compleja sintaxis de llamada al compilador o al linker.

8.2.- UBICACIÓN DE LA PILA

El linker L51, en ausencia de indicaciones en sentido contrario, asigna a la pila el mayor tamaño posible. Por ello, después de colocar los registros, las variables de tipo *bit*, la pila compilada, y las variables *data* e *idata*, posiciona el apuntador de pila (*stack pointer)* en la primera dirección de *idata* disponible. En caso de que se utilice un 8032, o cualquier otro derivado con 128 bytes de RAM interna por encima de la dirección 0x7f, también se puede utilizar este espacio para la pila.

```
?C_C51STARTUP   SEGMENT   CODE
?STACK          SEGMENT   IDATA ;Segmento en RAM de acceso indirecto

                RSEG      ?STACK    ; Reserva un byte
                DS        1
                EXTRN     CODE (?C_START)
                PUBLIC          ?C_STARTUP
                CSEG      AT    0
?C_STARTUP:     LJMP      STARTUP1

                RSEG      ?C_C51STARTUP
STARTUP1:
                MOV       SP,#?STACK-1 ; Apunta con SP a la pila
                LJMP      ?C_START     ; Salto a la sección de inicialización
```

8.3.- USO DE LOS 128 bytes SUPERIORES DE LA RAM DEL 8052

El µC 8051 dispone exclusivamente de 128 bytes de RAM interna, de acceso directo e indirecto. Los accesos directos a direcciones por encima de la 0x7f se refieren a los SFRs, y los accesos indirectos (MOV A,@R0) en esta región no funcionan.

Sin embargo, el µC 8052 y otros muchos derivados, disponen de 128 bytes por encima de la dirección 0x7f, que permiten el acceso indirecto. Para informar al linker de que se está utilizando un µC con 256 bytes de RAM interna se utiliza el comando **RAMSIZE(256)**. La principal aplicación de este espacio adicional es para la pila. El 8051 cuando empila un dato incrementa el registro apuntador SP (El 8086 lo decremeta), por esa razón el linker ubica la pila por encima del área utilizada por las variables. Si una aplicación no necesita una pila muy grande, se puede sacar partido a este espacio declarando algunas variables de tipo *idata*.

Muchos programadores no declaran variables de tipo *idata* hasta que otras variables han ocupado los 128 bytes inferiores. Sin embargo, es buena idea declarar de tipo *idata* algunos arrays y las variables a las que se accede mediante punteros, ya que en ambos casos el acceso indirecto (MOV A,@Ri) resulta ser el más apropiado. En estos casos, la pila se sitúa por encima de los objetos de tipo *idata*.

8.4.- SOLAPAMIENTO DE VARIABLES DATA REALIZADO POR L51

8.4.1.- Principios de solapamiento

Una de las principales características de C51 es la función de solapamiento, o mecanismo mediante el cual diferentes variables del programa utilizan las mismas posiciones de RAM interna. La posibilidad de utilizar solapamiento surge cuando se declaran variables automáticas. Estas variables tienen una vida muy corta, se crean al entrar a la función o bloque que las declara, y desaparecen después de la llave '}' de final de bloque o función. En consecuencia, el área ocupada por las variables automáticas queda libre y puede ser utilizada por otras funciones. Lo mismo sucede con el área de memoria utilizada por C51 para el paso de parámetros.

Si a lo largo de un programa, cada función conservara el área de memoria asignada a sus variables y parámetros, la RAM interna del 8051 se agotaría incluso con pequeños programas

En C51, el linker es quien se encarga de aplicar el mecanismo de solapamiento. El linker examina las necesidades de RAM interna de todas las funciones, y las llamadas que estas realizan entre sí, y con esa información determina los segmentos *data* y *bit* que pueden solaparse. El uso de los registros para ubicación de variables temporales también tiende a reducir las necesidades en el segmento *data*.

La función de solapamiento se basa en que si la función 1 llama a la función 2, sus áreas data no pueden solaparse, ya que ambas se encuentran activas al mismo tiempo. Una tercera función 3, también llamada por la 1, si puede solaparse con la 2, ya que las dos no pueden correr a la vez.

```
main
    |
   funcA - func2 - func3 - func4
    |
   funcB - func5 - func6 - func7
    |
   funcC - func8 - func9 - func10
    |
```

Como la función A llama a func2, y func2 llama a func3 etc., A, 2, 3 y 4 no pueden solapar sus áreas data. De igual forma B, 5, 6, y 7 tampoco pueden solaparse. Sin embargo los grupos 2,3,4; 5,6,7 y 8,9,10 si pueden solapar sus áreas data, ya que pertenecen a ramas distintas. Esta es la base de la estrategia de solapamiento.

Sin embargo las funciones de interrupción pueden producirse en cualquier momento, y si no se tomase un cuidado especial con ellas, podrían sobre-escribir las áreas *data* del programa principal (o de las interrupciones de menor prioridad). Para evitarlo, C51 identifica a las funciones de interrupción y a las funciones llamadas por estas, y las ubica en áreas de memoria individuales.

8.4.2.- Impacto del solapamiento en la construcción de programas

La regla general utilizada por L51 es que si dos funciones nunca se ejecutan a la vez, sus áreas data pueden solaparse. En el 99% de los casos el mecanismo de solapamiento trabaja correctamente, pero en algunas ocasiones pueden obtenerse resultados inesperados. Estos pocos casos son:

1. Las funciones llamadas indirectamente utilizando punteros
2. Las funciones llamadas desde tablas de llamada a funciones
3. Las funciones re-entrantes no declaradas de la forma apropiada

Cuando se dan estas condiciones el linker puede emitir los siguientes avisos:

```
MULTIPLE CALL TO SEGMENT
UNCALLED SEGMENT
RECURSIVE CALL TO SEGMENT
```

8.4.2.1.- Llamada indirecta a función mediante punteros

En el siguiente ejemplo, func4 y func5 se llaman desde main por medio de una función llamada EJECUTA que recibe como parámetro un puntero a la función a llamar. Cuando L51 analiza el programa, no es capaz de establecer un vínculo entre las funciones 4 y 5 porque el puntero a función que recibe como parámetro rompe la cadena de referencias (El valor del puntero a función es indeterminado durante el linkado). Por ello, L51 solapa los segmentos data de func4, func5 y ejecuta como si todas ellas fueran llamadas desde main.

Como resultado de lo anterior, las variables locales de func4 y func5 pueden destruir las variables locales de ejecuta, lo cual es MUY peligroso, especialmente cuando la sobre-escritura de variables no resulta obvia, y se produce solamente bajo condiciones anormales de operación. Ejemplo:

```
/*****************************************************************
*  Riesgo de solapamiento 1 - Funciones llamadas indirectamente  *
*****************************************************************/
#include<reg51.h>
char func1(void) {    // Función llamada directamente
 char x, arr[10] ;
for(x = 0 ; x < 10 ; x++) {
   arr[x] = x ;
   }
return(x) ;
}

char func2(void) {   // Función llamada directamente
//(.... Código C ...)
return 2;
}

char func3(void) {   // Función llamada directamente
//(.... Código C ...)
return(3);
}

char func4(void) {   // Función llamada indirectamente
char x4, arr4[10] ;  // variables locales
for(x4 = 0 ; x4 < 10 ; x4++) {
   arr4[x4] = x4 ;
   }
return(x4) ;
}

char func5(void) {   // Función llamada indirectamente
char x5, arr5[10] ;  // variables locales
```

```
for(x5 = 0 ; x5 < 10 ; x5++) {
   arr5[x5] = x5 ;
   }
return(x5) ;
}

/*** Función que llama a otras funciones ***/

char ejecuta(char (*fptr)())  //
   { // char (*fptr)(void) fptr es un puntero a función que retorna un
char
   char tex ;       // variables locales de ejecuta
   char arrex[10] ;
   for(tex = 0 ; tex < 10 ; tex++) {
      arrex[tex] = (*fptr)() ;
      }
   return(tex) ;
   }

/*** Declaración de array de punteros a función ***/

char (code *fp[3])(void) ;

/*** Llamadas a función desde main ***/

void main(void) {
   char am ;

   fp[0] = func1 ;     // Elementos del array de punteros a funciones
   fp[1] = func2 ;
   fp[2] = func3 ;

   am = (* fp[0])() ;        // Ejecución de las funciones
   am = (* fp[1])() ;
   am = (* fp[2])() ;

   if(P1) {            // Control para llamadas indirectas a función
      am = ejecuta(func4) ;
      }
   else {
      am = ejecuta(func5) ;
      }
   }
```

Resultados de condiciones peligrosas en el fichero '.M51' de salida del linker.

```
MS-DOS L51 LINKER/LOCATOR V3.70c, INVOKED BY:
E:\KEIL\BIN\L51.EXE MAIN.OBJ


MEMORY MODEL: SMALL


INPUT MODULES INCLUDED:
  MAIN.OBJ (MAIN)
  E:\KEIL\LIB\C51S.LIB (?C_STARTUP)
  E:\KEIL\LIB\C51S.LIB (?C?ICALL)


LINK MAP OF MODULE:  MAIN (MAIN)
```

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

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     000FH     UNIT         _DATA_GROUP_
            DATA    0017H     0006H     UNIT         ?DT?MAIN
            IDATA   001DH     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     0049H     UNIT         ?PR?MAIN?MAIN
            ......................................................


OVERLAY MAP OF MODULE:   MAIN (MAIN)


SEGMENT                            DATA_GROUP
  +--> CALLED SEGMENT            START    LENGTH
-----------------------------------------------
?C_C51STARTUP                    -----    -----
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                    0008H    0001H
  +--> ?PR?FUNC1?MAIN
  +--> ?PR?FUNC2?MAIN
  +--> ?PR?FUNC3?MAIN
  +--> ?PR?FUNC4?MAIN
  +--> ?PR?_EJECUTA?MAIN
  +--> ?PR?FUNC5?MAIN

?PR?FUNC1?MAIN                   0009H    000AH

?PR?FUNC4?MAIN                   0009H    000AH // Peligro de
solapamiento

?PR?_EJECUTA?MAIN                0009H    000EH // con FUNC4, EJECUTA y
FUNC5

?PR?FUNC5?MAIN                   0009H    000AH


SYMBOL TABLE OF MODULE:  MAIN (MAIN)

  VALUE           TYPE          NAME
  ----------------------------------

  -------         MODULE        MAIN
  D:0090H         PUBLIC        P1
  D:0017H         PUBLIC        FP

  -------         PROC          FUNC1
  D:0007H         SYMBOL        X
  D:0009H         SYMBOL        ARR
  -------         ENDPROC       FUNC1

  -------         PROC          FUNC4
  D:0007H         SYMBOL        X4
```

```
  D:0009H         SYMBOL        ARR4       // peligro
  -------         ENDPROC       FUNC4

  -------         PROC          FUNC5
  D:0007H         SYMBOL        X5
  D:0009H         SYMBOL        ARR5       // peligro
  -------         ENDPROC       FUNC5

  -------         PROC          _EJECUTA
  D:0009H         SYMBOL        FPTR       // peligro
  D:000CH         SYMBOL        TEX
  D:000DH         SYMBOL        ARREX
  -------         ENDPROC       _EJECUTA

  -------         PROC          MAIN
  D:0008H         SYMBOL        AM
  -------         ENDPROC       MAIN
LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

8.4.2.2.- Solución al caso de llamada indirecta a funciones

El problema señalado en el apartado anterior se soluciona mediante el uso del comando OVERLAY durante el proceso de linkado:

L51 main.obj OVERLAY(main ~ (func4,func5), _ejecuta ! (func4,func5))

Nota: El signo tilde '~' significa: "Ignorar las referencias a func4/5 desde main" El signo '!' significa: "Añadir a la lista las referencias entre la función 'ejecuta' y func4/5 para evitar el solapamiento de las variables locales de estas funciones."

La nueva salida del linker es:

```
MS-DOS L51 LINKER/LOCATOR V3.70c, INVOKED BY:
E:\KEIL\BIN\L51.EXE MAIN.OBJ OVERLAY (MAIN ~(FUNC4, FUNC5),_EJECUTA
!(FUNC4, FUNC5))


MEMORY MODEL: SMALL


INPUT MODULES INCLUDED:
  MAIN.OBJ (MAIN)
  E:\KEIL\LIB\C51S.LIB (?C_STARTUP)
  E:\KEIL\LIB\C51S.LIB (?C?ICALL)


LINK MAP OF MODULE:  MAIN (MAIN)


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

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0019H     UNIT         _DATA_GROUP_
            DATA    0021H     0006H     UNIT         ?DT?MAIN
            IDATA   0027H     0001H     UNIT         ?STACK
```

```
            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE


OVERLAY MAP OF MODULE:   MAIN (MAIN)


SEGMENT                            DATA_GROUP
  +--> CALLED SEGMENT            START    LENGTH
-----------------------------------------------
?C_C51STARTUP                    -----    -----
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                    0008H    0001H
  +--> ?PR?FUNC1?MAIN
  +--> ?PR?FUNC2?MAIN
  +--> ?PR?FUNC3?MAIN
  +--> ?PR?_EJECUTA?MAIN

?PR?FUNC1?MAIN                   0009H    000AH

?PR?_EJECUTA?MAIN                0009H    000EH
  +--> ?PR?FUNC4?MAIN
  +--> ?PR?FUNC5?MAIN

?PR?FUNC4?MAIN                   0017H    000AH

?PR?FUNC5?MAIN                   0017H    000AH



SYMBOL TABLE OF MODULE:  MAIN (MAIN)

  VALUE           TYPE          NAME
  ----------------------------------

  -------         MODULE        MAIN
  D:0090H         PUBLIC        P1
  D:0021H         PUBLIC        FP

  -------         PROC          FUNC1
  D:0007H         SYMBOL        X
  D:0009H         SYMBOL        ARR
  -------         ENDPROC       FUNC1


  -------         PROC          FUNC4
  D:0007H         SYMBOL        X4
  D:0017H         SYMBOL        ARR4
  -------         ENDPROC       FUNC4

  -------         PROC          FUNC5
  D:0007H         SYMBOL        X5
  D:0017H         SYMBOL        ARR5
  -------         ENDPROC       FUNC5

  -------         PROC          _EJECUTA
  D:0009H         SYMBOL        FPTR
  D:000CH         SYMBOL        TEX
  D:000DH         SYMBOL        ARREX
```

```
   -------         ENDPROC       _EJECUTA


   -------         PROC          MAIN
   D:0008H         SYMBOL        AM
   -------         ENDPROC       MAIN
   -------         ENDMOD        MAIN

LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

8.4.2.3.- Avisos en las llamadas a función mediante tablas

En el siguiente ejemplo se hacen llamadas a dos funciones por medio de un array de punteros a funciones. La tabla "tabla_saltos" existe en un segmento llamado "?CO?MAIN", es decir en el área de constantes asignada al módulo main. El problema surge debido a que los argumentos cadena de printf, también residen aquí. lo cual conduce una definición recursiva de la dirección de comienzo de las funciones en la tabla de saltos. Aunque ello no es peligroso, evita que se establezcan las referencias reales de las funciones y se inhibe el proceso de solapamiento.

```
#include <stdio.h>
#include <reg517.h>

void func1(void) {
   unsigned char i1 ;
   for(i1 = 0 ; i1 < 10 ; i1++) {
      printf("Esta es la función 1\n") ; // Cadena almacenada en
                                         // el segmento ?CO?MAIN
      }
   }

void func2(void) {
   unsigned char i2 ;
   for(i2 = 0 ; i2 < 10 ; i2++) {
      printf("Esta es la función 2\n") ; // Cadena almacenada en
                                         // el segmento ?CO?MAIN
      }
   }

code void(*tabla_saltos[])()={func1,func2}; // Tabla de saltos a
                                            // funciones, almacenada
                                           // en el segmento ?CO?MAIN
/*** Llamada a las funciones ***/

void main(void) {
   (*tabla_saltos[P1 & 0x01])() ;   // Llamada a función por medio
                                   // de la tabla en ?CO?MAIN
   }/* FIN DE main */
```

La salida '.MAP' del linker es:

```
MS-DOS L51 LINKER/LOCATOR V3.70c, INVOKED BY:
E:\KEIL\BIN\L51.EXE MAIN.OBJ


OVERLAY MAP OF MODULE:   MAIN (MAIN)
```

```
SEGMENT                            BIT_GROUP          DATA_GROUP
  +--> CALLED SEGMENT          START    LENGTH     START    LENGTH
-------------------------------------------------------------------
?C_C51STARTUP                  -----    -----      -----    -----
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                  -----    -----      -----    -----
  +--> ?CO?MAIN

?CO?MAIN                       -----    -----      -----    -----
  +--> ?PR?FUNC1?MAIN
  +--> ?PR?FUNC2?MAIN

?PR?FUNC1?MAIN                 -----    -----      0008H    0001H
  +--> ?PR?PRINTF?PRINTF


?PR?PRINTF?PRINTF              0020H.0  0001H.1    0009H    0014H
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?MAIN                 -----    -----      0008H    0001H
  +--> ?PR?PRINTF?PRINTF


*** WARNING 13: RECURSIVE CALL TO SEGMENT
    SEGMENT: ?CO?MAIN
    CALLER:  ?PR?FUNC1?MAIN

*** WARNING 13: RECURSIVE CALL TO SEGMENT
    SEGMENT: ?CO?MAIN
    CALLER:  ?PR?FUNC2?MAIN

LINK/LOCATE RUN COMPLETE.  2 WARNING(S),  0 ERROR(S)
```

8.4.24.- .Solución al caso de la tabla de saltos a funciones

La solución es usar el comando OVERLAY durante el linkado de la siguiente forma:

```
L51 main.obj OVERLAY(?CO?MAIN ~ (func1,func2), main ! (func1,func2))
```

La línea anterior borra las referencias desde ?CO?MAIN hacia func1 y func2, e inserta las verdaderas referencias desde main hacia func1 y func2. La salida que produce ahora el linker es:

```
MS-DOS L51 LINKER/LOCATOR V3.70c, INVOKED BY:
E:\KEIL\BIN\L51.EXE MAIN.OBJ OVERLAY(?CO?MAIN ~(FUNC1, FUNC2), MAIN
!(FUNC1, FUNC2))


OVERLAY MAP OF MODULE:   MAIN (MAIN)


SEGMENT                            BIT_GROUP          DATA_GROUP
  +--> CALLED SEGMENT          START    LENGTH     START    LENGTH
-------------------------------------------------------------------
?C_C51STARTUP                  -----    -----      -----    -----
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                  -----    -----      -----    -----
  +--> ?CO?MAIN
  +--> ?PR?FUNC1?MAIN
```

```
  +--> ?PR?FUNC2?MAIN

?PR?FUNC1?MAIN                 -----    -----      0008H    0001H
  +--> ?CO?MAIN
  +--> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF              0020H.0  0001H.1    0009H    0014H


  +--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?MAIN                 -----    -----      0008H    0001H
  +--> ?CO?MAIN
  +--> ?PR?PRINTF?PRINTF


LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

8.4.2.5.- Avisos de llamadas múltiples a segmentos

Este aviso sucede generalmente cuando se llama a una función desde el programa principal y desde una interrupción. Significa que potencialmente la interrupción puede llamar a una función que se encuentra corriendo por haber sido llamada desde el programa principal. La solución más sencilla es declarar a la función como REENTRANT para que el compilador genere una pila local para sus parámetros y variables. Así en cada llamada a la función se crea un nuevo conjunto parámetros y de variables locales sin destruir el conjunto existente.

Esta solución aumenta de forma significativa el tiempo de ejecución y el código generado. Otra posibilidad es la creación de una segunda versión de la función, una para el programa principal y otra para la interrupción, lo cual puede plantear un problema de mantenimiento, ya que se tienen dos versiones del mismo código.

En algunos casos la situación no es problemática, si el usuario ha procurado que el uso re-entrante de la función no suceda nunca, y se puede ignorar el aviso del linker. Esto puede lograrse permitiendo la interrupción solo cuando la llamada a la función sea segura. Sin embargo esta situación es peligrosa sobre todo si intervienen otros programadores. Ejemplo:

```
void func1(void) {
      unsigned char i1,a1[15] ;
      for(i1 = 0 ; i1 < 10 ; i1++) {
      a1[i1] = i1 ;
      }
   }

void func2(void) {
   // ( Código C de func2)
   }

void timer0_int(void) interrupt 1 {
   func1() ;
   }

main() {
   func1() ;
   func2() ;
   }/* FIN DE main */
```

El linker produce la siguiente salida:

```
OVERLAY MAP OF MODULE:   MAIN (MAIN)


SEGMENT                           DATA_GROUP
  +--> CALLED SEGMENT           START    LENGTH
-----------------------------------------------
?PR?TIMER0_INT?MAIN             -----    -----
  +--> ?PR?FUNC1?MAIN

?PR?FUNC1?MAIN                  0017H    000FH

*** NEW ROOT *************************************************

?C_C51STARTUP                   -----    -----
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                   -----    -----
  +--> ?PR?FUNC1?MAIN
  +--> ?PR?FUNC2?MAIN

*** WARNING 15: MULTIPLE CALL TO SEGMENT
    SEGMENT: ?PR?FUNC1?MAIN
    CALLER1: ?PR?TIMER0_INT?MAIN
    CALLER2: ?C_C51STARTUP

LINK/LOCATE RUN COMPLETE.  1 WARNING(S),  0 ERROR(S)
```

8.4.2.6.- Solución al caso de las llamadas múltiples a segmentos

Las posibles soluciones son:

(i) Declarar la función func1 como re-entrante:

```
void func1(void) reentrant {  }
```

(ii) Usar la opción OVERLAY del linker:

```
L51 main.obj OVERLAY(main ~ func1,timer0_int ~ func1)
```

para romper las conexiones entre func1 y las funciones main y timer0_int. En este segundo caso se obtiene la siguiente salida del linker:

```
OVERLAY MAP OF MODULE:   MAIN (MAIN)


SEGMENT
  +--> CALLED SEGMENT
---------------------
?C_C51STARTUP
  +--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN
  +--> ?PR?FUNC2?MAIN

*** WARNING 16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?FUNC1?MAIN

LINK/LOCATE RUN COMPLETE.  1 WARNING(S),  0 ERROR(S)
```

Lo que significa que no se producirá solapamiento entre func1 y otras funciones del programa principal. El aviso puede evitarse eliminando solamente el vínculo entre func1 y la interrupción:

```
L51 main.obj OVERLAY(timer0_int ~ func1)
```

Lo que produce la siguiente salida del linker:

```
OVERLAY MAP OF MODULE: MAIN (MAIN)


SEGMENT DATA_GROUP
+--> CALLED SEGMENT START LENGTH
------------------------------------------
?C_C51STARTUP ----- -----
+--> ?PR?MAIN?MAIN

?PR?MAIN?MAIN ----- -----
+--> ?PR?FUNC1?MAIN
+--> ?PR?FUNC2?MAIN

?PR?FUNC1?MAIN 0008H 000FH

LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)
```

También se puede deshabilitar totalmente el solapamiento, aunque se trata de una mala solución que puede agotar rápidamente la RAM interna:

```
main2.obj  to main2.abs  NOOVERLAY
```

Resumiendo, con el aviso "MULTIPLE CALL TO SEGMENT WARNING" la única solución realmente segura es declarar func1 como REENTRANT, quedando la opción de duplicar la función como la segunda mejor opción. El uso del comando OVERLAY corre el peligro de que un nuevo programador con menos experiencia no se de cuenta de que la interrupción se encuentra restringida a los instantes en que pueda llamar con seguridad a la función.

8.4.3.- Solapamiento de variables públicas

Los ejemplos anteriores se refieren exclusivamente al solapamiento de los parámetros y variables locales de las funciones. Otra situación diferente se planteó recientemente con un programa dividido en dos mitades. El 8051 debía de ejecutar una u otra mitad en función de la entrada que le proporcionara un usuario durante la fase de inicialización.

Cada mitad del programa tenía un gran número de variables públicas, algunas de las cuales eran comunes a las dos mitades del programa.

Este tipo de estructura de programa necesita una clase de almacenamiento nueva a la que llamaremos "PUBLICA" o conocida por ciertos módulos y que difiere de la clase de almacenamiento "GLOBAL" que resulta conocida por todos los módulos. El nuevo C166 soporta esta nueva clase de almacenamiento, pero no sucede así con el C51, por lo que se requiere algún tipo de ajuste.

El comando OVERLAY del linker no ayuda, ya que solo controla el solapamiento de los parámetros y variables locales de funciones. Una posible solución pasa por utilizar módulos especiales para declarar las variables públicas. Así el módulo 1 declara la variables públicas del

programa 1; el módulo 2 declara la variables públicas del programa 2; y finalmente el módulo 3 declara la variables comunes a ambos programas.

El truco consiste en utilizar el linker para ubicar los segmentos de datos de los módulos 1 y 2 en las mismas direcciones, mientras se permite que las variables del módulo 3 sean colocadas automáticamente por el linker. Esta solución usa tres módulos especiales para declarar las variables:

```
/* Ejemplo de creación de dos conjuntos de datos */
/* públicos en el mismo espacio de memoria */

extern void main1(void) ;
extern void main0(void) ;

/* Módulo principal que rompe el programa en dos partes */

void main(void) {
   bit flag ;
   if(flag) {
      main0() ;   // RAMA 0
      }
   else {
      main1() ;   // RAMA1 1
      }
   } ^^^^^^^^^^^^^^^^^^^^^^^ FINAL DEL MODULO PRINCIPAL

/* Módulo que declara las variables públicas para la RAMA 0 */
/* Públicos en la rama 0 */

unsigned char x2,y2 ;
unsigned int z2 ;
char a2[0x30] ;

/* Una variable accesible desde las dos ramas */
extern int comun ;

void main0(void) {
   unsigned char c0 ;
   x2 = 0x80 ;
   y2 = x2 ;
   c0 = y2 ;
   z2 = x2*y2 ;
   a2[2] = x2 ;
   comun = z2 ;
   }
^^^^^^^^^^^^^^^^^^^^^^^ FIN DEL MODULO


/* Módulo que declara las variables públicas para la RAMA 1 */
/* Públicos en la rama 1 */

unsigned char x1,y1 ;
unsigned int z1 ;
char a1[0x30] ;

/* Una variable accesible desde las dos ramas */
extern int comun ;

void main1(void) {
   char c1 ;
```

```
   x1 = 0x80 ;
   y1 = x1 ;
   c1 = y1 ;
   z1 = x1*y1 ;
   a1[2] = x1 ;
   comun = z1 ;
    }
^^^^^^^^^^^^^^^^^^^^^^^ FIN DEL MODULO


/* Módulo que declara las variables comunes a las dos RAMAS */

int comun ; /* Una variable accesible desde las dos ramas */

^^^^^^^^^^^^^^^^^^^^^^^ FIN DEL MODULO

/* Llamada al linker */
l51 t1.obj,t2.obj,com.obj to t.abs DATA(?DT?T1(20H),?DT?T2(20H))
```

La elección de la posición "20H" coloca los segmentos combinados justo encima de los bancos de registros.

El problema de esta solución es que el linker emite un aviso de solapamiento de segmentos data, que siempre resulta molesto, aunque en esta ocasión no resulte peligroso.

Capítulo 9 - Otras extensiones de C51

9.1.- FUNCIONES ESPECIALES PARA BITS

La necesidad de utilizar máscaras para comprobar el estado de un bit en char e ints, en las antiguas versiones de C51 resultaba frustrante para los programadores en lenguaje ensamblador, teniendo en cuenta que el 8051 dispone de un buen conjunto de instrucciones en ensamblador para acceder a bits individuales. Sin embargo, desde la versión 3 de C51, es posible hacer que determinadas variables residan en el área direccionable a nivel bit (D:0x20 a D:0x2F), donde es posible utilizar las instrucciones de bit del 8051.

Un ejemplo es la comprobación del signo de un char.

```
Aquí el char se declara como "bdata":
   char bdata test_char ;
   sbit sign_bit = test_char ^ 7 ;

para usarlo así:
   test_char = counter ;
   if(sign_bit==1) { /* test_char es negativo */ }
las instrucciones a ejecutar son:
   MOV   A,counter    ;
   MOV   test_char,A  ;
   JNB   0,HECHO      ;
     /* Negativo */
```

lo que resulta mucho más rápido que utilizar máscaras e instrucciones AND. La palabra *bdata* informa a C51 y L51 que la variable debe ubicarse en el área RAM direccionable como bit y como byte. La sentencia "sbit sign_bit = test_char ^ 7" le dice a C51 que el bit llamado sign_bit residirá en la posición 7 del byte test_char.

```
Número de Byte: test_char           20H    Inicio del área BDATA
Número de Bit:  0,1,2,3,4,5,6,7<_ sign_bit
Número de Byte:                     21H
Número de Bit:  8,9,10,11,12,13,14,15
Número de Byte:                     22H
Número de Bit:  16,17,18,19,20,21,22,23,24.....
```

La situación con los int es algo más complicada debido a que el 8051 no almacena los objetos como en principio se espera. El 8051 guarda el byte de mayor peso de un int en la dirección más baja. Por ello, el bit 7 es el bit de mayor peso del byte más significativo, y el bit 15 es el bit de mayor peso del byte menos significativo.

```
Número de byte: test_int(high)          20H
Número de bit:  0,1,2,3,4,5,6,7

Número de byte: test_int+1(low)         21H
Número de bit:  8,9,10,11,12,13,14,15
```

9.2.- SOPORTE PARA LA UNIDAD MATEMÁTICA DEL 80C517/537

Los Siemens 80C537 y 80C517A poseen una unidad matemática especial, la MDU, pensada para acelerar las aplicaciones que hagan un uso intensivo de las operaciones aritméticas con números.

9.2.1.- Cómo utilizar la MDU

Para facilitar la multiplicación y división de números de 16 y 32 bits (int y long), los Siemens 80C517 disponen de un co-procesador matemático (MDU) integrado en la CPU. También tiene operaciones de normalización y desplazamiento de 32 bits para soportar las operaciones con números en coma flotante. Además posee 8 registros apuntadores de datos (8 DPTRs) para facilitar el acceso a RAM externa.

El compilador puede sacar partido de estas mejoras del hardware si se utiliza el *switch* **"MOD517",** en la línea de comandos, o por medio de un #pragma. Así se hará uso de la MDU en las multiplicaciones y divisiones de más de 8 bit. Sin embargo el *linker* necesita una librería especial proporcionada por Keil.

El empleo de la MDU proporciona mejoras de 6 a 9 veces superiores a las del 8051 en la aritmética de enteros sin signo de 32 bits.

Una vez utilizado el *switch* **MOD517** se puede deshabilitar el uso de la MDU mediante los #pragmas **NOMDU** and **NODP**. La opción **NOMDU** inhibe el uso de la unidad matemática, mientras que **NODP** evita el uso de los 8 DPTRs.

9.2.2.- Los 8 apuntadores a datos

El 517A tiene 8 DPTRs que pueden acelerar el movimiento de bloques en memoria externa. El compilador C51 utiliza los DPTR únicamente con las funciones de librería memcpy() y strcpy().

Para utilizarlos el *switch* "MOD517" debe encontrarse activo. Notar que la función strcat() no utiliza los DPTR adicionales.

Si se utilizan los DPTR extra en el programa principal y en las funciones de interrupción, el registro DPSEL se almacena en la pila automáticamente al entrar en la interrupción utilizándose un nuevo valor en DPSEL mientras dure la función.

9.2.3.- Cosas a tener en cuenta con el 80C517

La MDU del 80C517 se usa como una subrutina hardware, ya que no forma parte de la CPU del 8051. Como tal, está sujeta a las reglas normales sobre re-entrancia de las subrutinas. Si por ejemplo, el programa principal y las rutinas de interrupción utilizan simultáneamente la MDU, los cálculos del programa principal serán erróneos. Esto se debe a que los registros de entrada y salida de la MDU ocupan posiciones fijas que pueden ser sobre-escritas por las rutinas de interrupción.

Para que el programa principal pueda detectar la sobre-escritura de los registros de la MDU, se utiliza el bit MDEF del registro ARCON. Si este bit se encuentra a "1" el cálculo debe repetirse. Otra solución consiste en hacer un uso apropiado del #pragma **NOMDU**.

Nota: el compilador no hace esto - el usuario debe añadir lo que sigue para superar el problema:

```
#pragma MOD517
#include "reg517.h"

    long x,y,z ;
    func()
      {
      while(1)
         {
           x = y / z ;       /* cálculo de 32-bit */
           if(MDEF == 0)     /* Repetir si los datos */
              { break ; }    /* han sido sobre-escritos */
         }                   /* Salir en caso contrario */
      }
```

9.3.- SOPORTE PARA EL 87C751

El Philips 87C751 se diferencia del 8051 en que solo dispone de 2k de memoria de código, sin que sea posible aumentar la ROM externamente. El compilador C51 puede ser forzado a utilizar las instrucciones de 2 bytes AJMP y ACALL en lugar de las instrucciones de 3 bytes LJMP y LCALL.

9.3.1.- 87C751 - Pasos a seguir

1. Invocar a C51 con C51 myfile.c ROM(SMALL) NOINTVECTOR o utilizar "#pragma ROM(SMALL)"
2. Utilizar el fichero startup INIT751.A51 del directorio LIB.
3. No utilizar aritmética de coma flotante, divisiones de int o long, printf, scanf etc., ya que utilizan LCALLs.
4. Se dispone de una librería especial para el 87C751 que contiene llamadas cortas a las funciones de la librería estándar.

9.3.2.- Promoción de enteros

Para cumplir con las recientes exigencias ANSI, a partir de la versión 3.40 de C51, las sentencias IF incorporan la promoción de enteros, lo cual facilita la portabilidad desde los compiladores C de Microsoft o Borland. Así, cualquier char dentro de una sentencia condicional se convierte a int antes de realizar cualquier comparación. Esta característica tiene sentido en máquinas de 16 bits que son tan eficientes o más con los int, que con los char, pero produce una pérdida de eficiencia en el 8051. Por ello, Keil proporciona el "#pragma NOINTPROMOTE" para deshabilitar la promoción a enteros. En este caso se puede utilizar un forzado explícito (cast) si otro tipo de dato resulta afectado por la operación.

```
Este fragmento de C demuestra la importancia de este #pragma:
char c ;
unsigned char c1, c2 ;
int i ;
main() {
   if((char)c == 0xff) c = 0 ;
   if((char)c == -1) c = 1 ;
   i = (char)c + 5 ;
   if((char)c1 < (char)c2 + 4) c1 = 0 ;
```

```
}
```

Tamaños de código

```
47 bytes - C51 v3.20
76 bytes - C51 v5.50 (INTPROMOTE)
53 bytes - C51 v5.50 (NOINTPROMOTE)
```

Una vez más se demuestra que la portabilidad compromete la eficiencia en los programas del 8051 ...

Capítulo 10 - Miscelanea de puntos

10.1.- VINCULACIÓN DEL PROGRAMA C AL VECTOR RESTART

La vinculación del programa C al vector *Restart* se logra mediante el fichero en ensamblador STARTUP.A51. La rutina startup inicializa las variables en RAM, la pila y ejecuta un LJMP a "main", la primera función de un programa C.

10.2.- FUNCIONES INTRÍNSECAS

Hay un número de instrucciones especiales del 8051 que normalmente no utiliza el compilador C51. Por razones de velocidad, a veces resulta útil tener acceso a las mismas.

A diferencia del operador ">>", la función _cror_ permite utilizar la instrucción del 8051 "RR A" (*rotate accumulator right*) para rotar un char. Esto proporciona un resultado mucho más rápido que el que puede obtenerse mediante el operador >>. De igual forma las funciones intrínsecas _iror_ y _lror_ sirven para rotar a la derecha valores de tipo int y long respectivamente.

La función _nop_ añade simplemente una instrucción NOP para generar un retardo de tiempo corto y predecible. Otra función _testbit_, utiliza la instrucción JBC para comprobar el estado de un bit y en caso de que se encuentre a "1", borrarlo y saltar a otra región del programa.

Las funciones intrínsecas incluyen su código en el lugar donde se las nombra, eliminando las instrucciones de llamada y retorno que siempre están presentes en las llamadas normales a funciones. Para hacer uso de las mismas se necesita incluir el fichero "intrins.h" en el fichero fuente.

El siguiente ejemplo utiliza la función intrínseca _testbit_() ahorrando una instrucción CLR:

```
; #include <intrins.h>
;
;
; unsigned int shift_reg = 0 ;
;
; bit test_flag ;
;
; void main(void) {
    RSEG  ?PR?main?T
    USING    0
main:
            ; SOURCE LINE # 12
;
; /* Utilizando la forma normal */
;
;    test_flag = 1 ;
            ; SOURCE LINE # 14
    SETB     test_flag
;
;    if(test_flag == 1) {
            ; SOURCE LINE # 16
    JNB      test_flag,?C0001
;       test_flag = 0 ;
            ; SOURCE LINE # 17
    CLR      test_flag
```

```
;       P1 = 0xff     ;
            ; SOURCE LINE # 18
    MOV      P1,#0FFH
;       }
            ; SOURCE LINE # 19
?C0001:
;
; /* Utilizando una función intrínseca */
;
;    test_flag = 1 ;
            ; SOURCE LINE # 21
    SETB     test_flag
;
;   if(!_testbit_(test_flag)) {
            ; SOURCE LINE # 23
    JBC      test_flag,?C0003
;       P1 = 0xff     ;
            ; SOURCE LINE # 24
    MOV      P1,#0FFH
;       }            ; SOURCE LINE # 25
;
;    }
            ; SOURCE LINE # 27
?C0003:
    RET
; END OF main
    END
```

10.3.- CONTROL #pragma DEL BIT EA

La directiva DISABLE impide las interrupciones durante la duración de una función. Cada función que tenga que ser ejecutada con las interrupciones deshabilitadas debe llevar un "#pragma disable" antes de la definición de la función.

10.4.- SOPORTE PARA SFR DE 16 bits

Dentro del gran número de derivados del 8051, hay microcontroladores que poseen registros de 16 bits, como los registros de captura y comparación. El tipo sfr16 permite tratar desde C, a estos registros de 16 bits como un todo, sin necesidad de tratar los bytes altos y bajos individualmente. Para poder utilizar esta característica, es necesario que los registros de 16 bits ocupen posiciones consecutivas en el área de los SFR. En la declaración de los sfr de 16 bits se utiliza siempre el byte bajo del mismo. Por ejemplo, sabiendo que el sfr DPL ocupa la dirección 0x82 y que el sfr DPH ocupa la dirección 0x83, se puede declarar un registro de 16 bits así "sfr16 DPTR = 0x82;". Notar que los *timers* T0 y T1 del 8051 no tienen sus bytes bajos y altos dispuestos secuencialmente y no pueden hacer uso de esta característica. En cualquier caso, la declaración de registros de 16 bits facilita su tratamiento desde C, aunque necesariamente el 8051 utiliza instrucciones de 8 bits.

10.5.- NIVELES DE OPTIMIZACIÓN DE FUNCIONES

Los niveles de optimización de funciones por encima del 4, exigen que toda la función resida en la memoria del PC durante el proceso. En caso de que la memoria resulte insuficiente, se emite un aviso (*warning*) y se abandona el proceso de optimización.

10.6.- FUNCIONES In-line EN C51

Uno de los fundamentos de C es que las funciones tienen bien definido el mecanismo de entrada y salida. Esto significa que los parámetros de entrada se colocan en un área definida, en la pila o en los registros, y luego se ejecuta un CALL. Inevitablemente, la instrucción de llamada guarda dos bytes en la pila (la dirección de retorno).

En la mayoría de las aplicaciones del 8051, esto no es ningún problema, ya que generalmente se dispone de 256 bytes de RAM interna para la pila, lo que permite funciones con un elevado nivel de anidamiento.

Sin embargo en el caso del 8031 y de un reducido número de dispositivos tales como el 87C751, cada byte de RAM interna es crítico. En el último caso solo se dispone de 64 bytes.

Un truco que permite salvar espacio en la pila y reducir el tiempo de ejecución es utilizar macros con parámetros que actúan como funciones "*in-line*". Aunque el uso de macros con parámetros no es una práctica habitual, puede resultar muy útil en variantes del 8051 con limitada RAM interna.

En el siguiente ejemplo, el trabajo de la función strcpy() lo realiza una macro llamada "Inline_Strcpy", que aunque parece una función normal, no posee una dirección fija ni un área para datos propia. El carácter '\' sirve para que la definición de la macro pueda continuar en varias líneas.

Estas funciones se llaman como las funciones normales, con los parámetros metidos dentro de ( ). Sin embargo no se utiliza un CALL porque el código necesario se crea en el punto de llamada "*in-line*". El resultado final es que se realiza el trabajo de strcpy pero sin necesidad de usar la pila.

La desventaja de este método, en este ejemplo concreto, es que los apuntadores a la fuente y al destino se modifican durante el proceso de copia.

Un beneficio adicional en este ejemplo es que los punteros s1 y s2, son punteros específicos, y por lo tanto muy eficientes. Los lentos punteros genéricos no son necesarios aunque haya que copiar datos en distintas áreas de memoria.

```
#define Inline_Strcpy(s1,s2)  { while((*s1 = *s2) != 0))\
                                  {*s1++ ; *s2++; }\
                                }
char xdata *out_buffx = "                              " ;
char xdata *in_buffx  = "Hello" ;
char idata *in_buffi  = "Hello" ;
char idata *out_buffi = "                              " ;
char code *in_buffc   = "Hello" ;

void main(void) {

   Inline_Strcpy(out_buffx,in_buffx)  /* funciones In-line */
   Inline_Strcpy(out_buffi,in_buffi)
   Inline_Strcpy(out_buffx,in_buffc)
   }
```

El cálculo de la interpolación realizado originalmente por una subrutina, puede redefinirse fácilmente como una macro de 5 parámetros, minimizando el uso de RAM y el tiempo de

ejecución a costa del tamaño del código. Notar que 'r', el quinto parámetro, representa el valor de retorno que hay que pasar a la macro, para que pueda poner el resultado en algún sitio.

```
#define UCHAR unsigned char
#define UINT  unsigned int

#define interp_sub(x,y,n,d,r)  y -= x ; \
if(!CY) { r = (UCHAR) (x + (UCHAR)(((UINT)(n *  y))/d)) ; }\
else    { r = (UCHAR) (x - (UCHAR)(((UINT)(n * -y))/d)) ; }

Esta macro se llama en:

/*Interpolar valores 2D */
/*Uso de macro con parámetros*/

interp_sub(map_x1y1,map_x2y1,x_temp1,x_temp2,result_y1)

más tarde se reutiliza con otros parámetros así:

interp_sub(map_x1y2,map_x2y2,x_temp1,x_temp2,result_y2)
```

Para resumir, las macros con parámetros proporcionan un buen mecanismo para ordenar a C51 la ejecución de operaciones con distintos valores de entrada, en programas en los que la velocidad o el espacio en RAM es crítico.

Capítulo 11 - Algunos trucos de programación con C51

11.1.- ACCESO A R0 etc. DIRECTAMENTE DESDE C51

Un usuario de C51 siguió utilizando rutinas existentes en ensamblador para realizar tareas específicas. Por razones históricas las rutinas en ensamblador retornaban valores de 8 bits en el registro R0 del banco 3. Sin embargo C51 retorna valores *char* en R7, por ello la asignación a una variable del valor devuelto por la llamada a una función en ensamblador no funcionaría.

La solución consistió en declarar un puntero específico al área de memoria DATA. En tiempo de ejecución se asignó al puntero la dirección absoluta del registro (0x18). El valor de retorno de la función en ensamblador se recogió mediante este puntero.

```
/*** Ejemplo de acceso a registros específicos en C ***/
char data *dptr ;  /* Crear un puntero a DATA */

/* Definir la dirección del registro */

#define R0_bank3 0x40018L   /* Dirección de R0 en */
                            /* banco 3, 4 => espacio DATA */
char x,y ;

/* Ejecución */

main() {
dptr = (char*) R0_bank3 ;  /* Apunta a R0, banco3 */

x = 10 ;
dptr[0] = x ;   /* Escribe x en R0, banco3 */
y = *dptr ;     /* Lee el valor de R0, banco3 */

}
```

Otra alternativa podría consistir en declarar una variable para almacenar el valor de retorno, en un modulo separado y utilizar el *linker* para ajustar ese módulo a la dirección 0x18 del segmento DATA. Este método es más robusto y eficiente, pero considerablemente menos flexible.

11.2.- USO DE LAS FUENTES DE INTERRUPCIÓN SOBRANTES

Un problema del 8051 es la carencia de un TRAP o interrupción software. Mientras que los usuarios de C166 tienen el lujo de disponer de un hardware que soporta tales cosas, los programadores del 8051 tienen que ser más ingeniosos.

Recientemente se planteó un problema en el cual, la función de interrupción de mayor prioridad tenía que correr hasta cierto punto, a partir del cual otras interrupciones podrían entrar. Desafortunadamente no sirve cambiar el registro de prioridad de las interrupciones durante la propia función de interrupción, debido a que las interrupciones de menor o igual prioridad quedan a la espera del RETI. La solución consistió en utilizar la interrupción del convertidor A/D, que estaba libre, y encadenar la segunda sección de la función de interrupción a ella. Para ello, se activó el *flag* IADC de interrupción pendiente, justo antes de la llave de cierre "}", con lo cual la segunda sección de la interrupción podía correr inmediatamente. Como la prioridad de la interrupción del ADC se había puesto a nivel bajo, podía ser interrumpida por interrupciones de nivel alto.

```
/* Interrupción primaria de la entrada capture CC0 */

tdc_int() interrupt 8 {

/* Sección de alta prioridad - no debe ser interrumpida */


/* Habilitar la sección de menor prioridad mediante */
                         /* la interrupción del ADC */

IADC = 1 ; /* Solicita la interrupción del ADC */
EADC = 1 ; /* Habilita la interrupción del ADC */
}

/* Sección de menor prioridad unida a la interrupción del ADC */

tdc_int_low_priority() interrupt 10

IADC = 0 ; /* Evita nuevas llamadas */
EADC = 0 ;

/* La sección de baja prioridad debe ser interrumpible y */
/* debe seguir a la sección de alta prioridad de arriba */

}
```

11.3.- DISPOSITIVO DE CONMUTACIÓN DE MEMORIA DE CÓDIGO

Este truco se utilizó durante el desarrollo de un módulo cargador de ficheros HEX para un monitor sencillo de 8051. Tras recibir un fichero HEX en RAM, a través del puerto serie, debía ejecutarse el programa recibido comenzando en la dirección 0000H. La complicación estaba en que había que conmutar la memoria antes de ejecutar el nuevo programa.

La solución consistió en colocar la sección de conmutación de memoria en la dirección 0xFFFD para que la búsqueda de la siguiente instrucción se realizara en la dirección 0x0000, simulando de esta forma un reset. Idealmente todos los registros y *flags* deberían haber sido borrados previamente.

```
#include "reg.h"
#include "cemb537.h"
#include <stdio.h>

void main(void)
   {
   unsigned char tx_char,rx_char,i ;
   P4 = map2 ;
   v24ini_537() ;
   timer0_init_537() ;
   hexload_ini() ;
   EAL = 1 ;
   while(download_completed == 0)
      {
      while(char_received_fl == 0)
         { receive_byte() ; }
      tx_byte = rx_byte ; /* Eco */
      hexload() ;
```

```
      send_byte(tx_byte) ;
      char_received_fl = 0 ;
      }
   real_time_count = 0 ;
   while(real_time_count < 200)
      { ; }
   i = ((unsigned char (code*)(void)) 0xFFFD) () ;
       /* Salto a la dirección absoluta 0xfffd. */
   }
//^^^^^^^^^^^^^^^^^^^^^^^^ Final del Módulo


;
   NAME SWITCH
;
; Hace que PC rebase la dirección 0xFFFF simulando un reset
;
   P4      DATA 0E8H
;
   CSEG AT 0FFFDH
;
   MOV  P4,#02Fh  ;
;
   END

//^^^^^^^^^^^^^^^^^^^^^^^^ Final del Módulo "MAPCON"
```

Hay otras formas de hacer esto. Por ejemplo, el código del módulo MAPCON podría ubicarse durante el *linkado* así: CODE(SWITCH(0FFFDH)), evitando el uso de "CSEG AT".

11.4.- SIMULACIÓN DE UN RESET SOFTWARE

El 8051, a diferencia del 80C166 no posee una instrucción de reset. Este método utiliza un puntero a función para realizar la llamada a la dirección 0, y simular un reset.

Sin embargo, debe recordarse que para que el reset pueda ser considerado como tal, los registros de la CPU deberían quedar en el mismo estado que tras un reset real, y la dirección de retorno almacenada en la pila debería ser extraída igualmente.

```
; void main(void) {

    RSEG  ?PR?main?T1
    USING    0
main:
;
; ((void (code*) (void)) 0x0000) () ;
    LCALL    00H       ; Salto a la dirección CERO!
;
; }
    RET
; END OF main
```

11.5.- LA DIRECTIVA DEL PREPROCESADOR - #define

Se trata de un mecanismo para reemplazar texto, que puede utilizarse para facilitar la lectura de los programas dando nombres significativos a las constantes, por ejemplo:.

```
#define fuel_constant 100 * 2
```

así la sentencia "temp = fuel_constant" asignará el valor 200 a temp.

Anótese que el preprocesador solo permite cálculos con enteros.

Capítulo 12 - Funciones de libreria C51

12.1.- INTRODUCCIÓN

Una de las principales características de C es su habilidad para construir funciones complejas utilizando comandos básicos. Para aliviar el esfuerzo del programador, el compilador se suministra con muchas funciones matemáticas y de manejo de cadenas. Estas funciones se encuentran ya compiladas en ficheros de librería.

12.2.- LLAMADA A FUNCIONES DE LIBRERÍA

Las funciones de librería se llaman igual que las funciones definidas por el usuario.

```
#include ctype.h
{
char test_byte ;
result = isdigit(test_byte) ;
}
```

donde "isdigit( )" es una función que devuelve valor 1 (*true*) si test_byte es un carácter ASCII en la gama 0 a 9.

Las declaraciones de las funciones de librería se encuentran en ficheros con extensión ".h" (Ficheros de cabecera)- Véase el fragmento de código anterior.

Otros ejemplos son:

```
ctype.h,
stdio.h,
string.h etc..
```

Cuando en un fichero se utilicen funciones de librerías se debe incluir el o los ficheros cabecera implicados a comienzo del mismo.

Las funciones matemáticas disponibles, tales como ln, log, exp, 10x, sin, cos, tan (y las hiperbólicas equivalentes), operan con números de coma flotante (*float*). El fichero cabecera que contiene los prototipos de las funciones matemáticas es "math.h".

Los ficheros librería contienen muchas funciones que pueden utilizarse en los programas C. Las funciones de librería implicadas en un programa las extrae el linker (Capítulo 8) de la librería. Estos ficheros se tratan como librerías en función de su estructura y no en función de su extensión. Es posible eliminar o añadir módulos a una librería por medio del programa gestor de librerías LIB51.

12.3.- LIBRERÍAS ESPECÍFICAS A CADA MODELO DE MEMORIA

Cada modelo de memoria requiere una librería diferenciada. Obviamente si se utiliza el modelo LARGE en un programa, el código utilizado deberá ser distinto al utilizado cuando el modelo de memoria sea SMALL.

Así el compilador C51, se suministra con 6 librerías distintas, tres para los tres modelos de memoria y otras tres que incluyen funciones para operaciones con variables de tipo float:

```
C51S.LIB   - modelo SMALL
C51C.LIB   - modelo COMPACT
C51L.LIB   - modelo LARGE
C51FPS.LIB - modelo SMALL coma flotante (float point)
C51FPC.LIB - modelo COMPACT coma flotante
C51FPL.LIB - modelo LARGE coma flotante
```

Las funciones de librería de C51 son independientes del banco de registros utilizado por la aplicación. Esto significa que las funciones de librería pueden utilizarse sin tener en cuenta el estado actual de REGISTERBANK( ) o USING. Esta es una gran ventaja, ya que se pueden utilizar funciones de librería en el programa principal y en las funciones de interrupción sin preocuparse del banco de registros que se encuentre activo.

Capítulo 13 - Ficheros de salida del compilador C51

13.1.- FICHEROS OBJETO

Al estar C51 íntimamente relacionado con las herramientas originales de Intel, los ficheros objeto creados por C51 se encuentran en formato Intel (*Intel object file format*). Este formato corresponde a un fichero binario que contiene la información simbólica necesaria para la depuración con emuladores. Los ficheros en este formato pueden unirse con ficheros objeto generados por Intel PLM51 or ASM51 usando el *linker* Keil L51. La salida final es Intel OMF51.

Las versiones del compilador superiores a la 2.3 producen un fichero Intel OMF51 extendido cuando se utilizan los *switches* DEBUG OBJECTEXTEND. Así se pasa información de tipo y de alcance de las variables que puede utilizarse con depuradores y emuladores. Las extensiones al formato original Intel constituyen un desarrollo propiedad de Keil que ha sido copiado por IAR y otros.

13.2.- FICHEROS HEX PARA GRABADO DE EPROMs

Si se desea programar una EPROM se necesita una etapa adicional que genere un fichero HEX. Los ficheros en formato Intel HEX son representaciones ASCII de programas absolutos sin información sobre símbolos. Todos los equipos programadores de EPROM aceptan ficheros Intel HEX. La utilidad OH51 se utiliza para convertir el fichero OMF51, generado por el *linker*, al formato estándar de 8 bits Intel HEX.

13.3.- FICHERO EN LENGUAJE ENSAMBLADOR

Por medio del *switch* SRC en la línea de comandos, o incluyendo en la primera línea de un fichero la sentencia "#pragma SRC", el compilador C51 crea un fichero con extensión ".src" que contiene las líneas de código fuente originales C, como líneas de comentarios, junto a las líneas en lenguaje ensamblador. Esta opción es muy útil para saber cómo controla el compilador al 8051.

Normalmente no es conveniente cambiar manualmente la salida del compilador, ya que pueden crearse programas muy difíciles de mantener. Es preferible estructurar el código fuente para escribir código eficiente desde el principio. Los programas sencillos y eficientes en C producen el mejor código para el 8051.

Capítulo 14 - LLamadas a funciones en ensamblador

14.1.- EJEMPLO DE FUNCIÓN EN ENSAMBLADOR

No es difícil llamar a rutinas en ensamblador desde C51, si se lee este capítulo y el manual del usuario del compilador.

En contadas ocasiones el código generado por el compilador C51 no es adecuado para una determinada aplicación y se necesita recurrir al lenguaje ensamblador.

Si una función escrita en lenguaje ensamblador no recibe parámetros, la llamada a la misma es como la llamada a cualquier función C. Se necesita un prototipo de función *extern* antes de realizar una llamada a la misma:

Fichero C51:

```
extern void asm_func(void).
```

Fichero A51:

```
ASM_FUNC:  MOV  A,#10   ; instrucciones en ensamblador 8051
```

Si se deben pasar parámetros, C51 colocará los primeros parámetros en registros, tal como se apunta en esta sección.

La complicación aparece cuando todos los parámetros no caben en los registros. En este caso el usuario debe declarar un área de memoria en la que poder ubicar los parámetros extra. Así la función en ensamblador debe tener definido un segmento DATA conforme a los convenios sobre nombres esperados por C51.

En el ejemplo que sigue, el segmento

```
?DT?_WRITE_EE_PAGE?WRITE_EE SEGMENT DATA OVERLAYABLE
```

hace exactamente eso.

El mejor consejo es escribir un programa C que llame a la función en ensamblador, y compilarlo con la opción SRC para producir su equivalente en ensamblador. Luego debe observarse lo que hace C51 cuando llama la la función en ensamblador todavía sin escribir. Si se utiliza el nombre de segmento generado por C51 no habrá problemas.

Ejemplo de función en ensamblador con muchos parámetros

Llamada a función desde C

Las siguientes líneas deben añadirse al programa C que llama a la función en ensamblador:

```
#define UCHAR unsigned char
/* referencia "extern" a la rutina en ensamblador
*/
extern UCHAR write_ee_page(char*,UCHAR,UCHAR) ;
```

```
void dummy(void)
  {
  UCHAR number, eeprom_page_buffer,
ee_page_length ;
  char * current_ee_page ;

  number = write_ee_page (current_ee_page,
       eeprom_page_buffer, ee_page_length) ;
  } /* FIN DE dummy */
```

La rutina en ensamblador es:

```
 NAME EEPROM_WRITE ;


    PUBLIC  _WRITE_EE_PAGE              ; Esencial!!!
    PUBLIC  ?_WRITE_EE_PAGE?END_ADDRESS ;
    PUBLIC  ?_WRITE_EE_PAGE?END_BUFFER  ;
;
P6     EQU  0FAH  ;
Un pin del puerto P6 actúa sobre el watchdog
;*************************************************************
;*<<<<<<<<< Declarar los segmentos CODE y DATA para la
           Rutina en Ensamblador >>>>>>>>>>>*
;*************************************************************;
?PR?_WRITE_EE_PAGE?WRITE_EE SEGMENT CODE
?DT?_WRITE_EE_PAGE?WRITE_EE SEGMENT DATA OVERLAYABLE ;
;
;*************************************************************
;*<<<<<< Declarar el área de memoria en RAM interna para
       Variables locales, Etc. >>>>>>*
;*************************************************************
;
        RSEG ?DT?_WRITE_EE_PAGE?WRITE;
?_WRITE_EE_PAGE?END_ADDRESS:  DS   2   ;
?_WRITE_EE_PAGE?END_BUFFER:   DS   1   ;
;
;
;*************************************************************
;*<<<<<<<<<<<<<<     Función de escritura de página EEPROM
;*************************************************************
;
        RSEG   ?PR?_WRITE_EE_PAGE?WRITE ;
; _
WRITE_EE_PAGE:
        CLR    EA
        MOV    DPH,R6  ; Dirección de la EEPROM en R7/R6
        MOV    DPL,R7  ;
;
        MOV    A,R3  ; Longitud del buffer en R3
        DEC    A     ;
        ADD    A,R7  ; Calcular dirección del último byte de la
        MOV    ?_WRITE_EE_PAGE?END_ADDRESS+01H,A ; página en XDATA.
        CLR    A                                 ;
        ADDC   A,R6                              ;
        MOV    ?_WRITE_EE_PAGE?END_ADDRESS,A     ;
;
        MOV    A,R5   ; La dirección del buffer IDATA en R5
        MOV    R0,A   ;
        ADD    A,R3     ;
```

```
        MOV     ?_WRITE_EE_PAGE?END_BUFFER,A ;
;
LOOP:   MOV     A,@R0       ;
        MOVX    @DPTR,A     ;
        INC     R0          ;
        INC     DPTR        ;
        MOV     A,R0        ;
        CJNE    A,?_WRITE_EE_PAGE?END_BUFFER,LOOP ;
;
        MOV     DPH,?_WRITE_EE_PAGE?END_ADDRESS      ;
        MOV     DPL,?_WRITE_EE_PAGE?END_ADDRESS+01H  ;
        DEC     R0          ;
;
CHECK:  XRL     P6,#08      ; Refrescar el watchdog del MAX691
        MOVX    A,@DPTR     ;
        CLR     C           ;
        SUBB    A,@R0       ;
        JNZ     CHECK       ;
;
        SETB    EA          ;
        RET                 ; Retorna al programa C
;
        END
;
```

14.2.- PASO DE PARÁMETROS A FUNCIONES EN ENSAMBLADOR

En el ejemplo anterior, el parámetro current_ee_page se recibió en R6 y R7 con el byte más significativo en el registro de número inferior R6. El que el 8051 almacene los bytes altos de los objetos multibyte, en las direcciones bajas es la causa de muchos quebraderos de cabeza.

Desde la versión 3.0 de C51, el prefijo "_" en la función en ensamblador WRITE_EE_PAGE es un convenio para indicar que se utilizan los registros para el paso de parámetros.

Cuando se pasan más parámetros de los que los registros pueden almacenar, se toma espacio en el área de memoria correspondiente al modelo utilizado (SMALL-*data*, COMPACT-*pdata*, LARGE-*xdata*).

14.3.- PASO DE PARÁMETROS EN REGISTROS

Las versiones modernas de C51 admiten el paso de parámetros a través de los registros de la CPU (R0-R7), lo cual permite realizar llamadas muy rápidas a funciones. En los registros se pueden pasar hasta tres parámetros, pero si alguno de ellos es *long* o *float*, sólo se podrán pasar dos parámetros ya que un *long* o *float* ocupa 4 bytes y únicamente hay 8 registros disponibles. Para mantener la compatibilidad con las versiones antiguas de C51 se utiliza la sentencia "#pragma NOREGPARMS que obliga a utilizar posiciones fijas de memoria para el paso de parámetros.

Tipo Parámetro	char	int	long/float	Generic Ptr
Parámetro 1°	R7	R6/R7	R4-R7	R1,R2,R3
Parámetro 2°	R5	R4/R5	R4-R7	R1,R2,R3
Parámetro 3°	R3	R2/R3		R1,R2,R3

```
Los punteros específicos de 1 byte y 2 bytes se pasan igual que los
char
y los int respectivamente
```

Capítulo 15 - Reglas generales a seguir

15.1.- REGLAS GENERALES A SEGUIR

Las siguientes reglas permiten que el compilador haga el mejor uso de los recursos del procesador. En general, la aproximación a C desde el punto de vista del programador en ensamblador no es dañina en absoluto.

Regla 1

Utilizar siempre que sea posible variables de 8 bit. El 8051 es una máquina de 8 bits que procesa los char con mayor eficacia que los int.

Regla 2

Utilizar siempre que sea posible variables unsigned. El 8051 no tiene instrucciones para las operaciones aritméticas con signo, por lo cual las operaciones con signo siempre añaden más instrucciones del 8051.

Regla 3

Procurar eliminar las divisiones salvo que sean entre números de 8 bits. El 8051 tiene una sola instrucción para dividir dos números de 8 bits. Dividir números de 32 bits entre números de 16 bits puede resultar muy lento, salvo que se utilice un 80C537.

Regla 4

Evitar el uso de estructuras de bit que producen código lento e ineficaz. En su lugar declarar bits individualmente, utilizando la clase de almacenamiento "*bit*".

Regla 5

El estándar ANSI dice que el producto de dos cantidades de 8 bits (char) es también un char. En consecuencia, cualquier unsigned char que al ser multiplicado pueda producir un resultado superior a 255 debe declararse como unsigned int.

Pero tal como señala la regla 15.1 no debe utilizarse un int, cuando pueda servir un char. La solución es convertir temporalmente (*cast*) el char a int. En el siguiente ejemplo el producto potencialmente puede tener 16 bits, pero el resultado es siempre de 8 bits. El *cast* o forzado de tipo "(unsigned int)" asegura que el C51 realice una multiplicación de 16 bits.

```
{
unsigned char z ;
unsigned char x ;
unsigned char y ;
z = ((unsigned int) y * (unsigned int) x) >> 8 ;
}
```

Aquí se multiplican dos números de 8 bits, y el resultado se divide entre 256. El resultado intermedio tiene 16 bits debido a que los números x e y, han sido cargados por la rutina de multiplicación de la librería como ints.

Regla 6

Los cálculos con operandos enteros que, debido a un cuidadoso escalado, siempre producen resultados de 8 bits siempre funcionarán. En:

```
unsigned int x, y ;
unsigned char z ;
z = x*y/256 ;
```

C51 igualará z al byte de dirección más alta (menos significativo) del resultado entero. Este resultado es independiente de la máquina utilizada, ya que viene impuesto por el estándar ANSI. En este caso C51 accede al byte menos significativo directamente, ahorrando código, ya que la división no se realiza.

15.1.1.- Números en coma flotante

En las operaciones con números en coma flotante, un operando siempre se mete en la pila aritmética de RAM interna. En el modelo SMALL se usa la pila del 8051, pero en los otros modelos se crea un segmento fijo en la primera dirección disponible por encima del área del banco de registros. En las aplicaciones en las que sea necesario ahorrar espacio en RAM interna, no debería utilizarse matemática de coma flotante. La coma fija es una alternativa más real.

Capitulo 16. Conclusión

Lo visto en este tema habrá proporcionado una ligera idea de cómo el compilador C51 puede utilizarse en el desarrollo de programas reales. Su gran ventaja es que elimina la necesidad de ser un experto en ensamblador del 8051 para producir programas eficaces.

Realmente, C51 puede considerarse como un lenguaje universal de nivel entre medio y bajo, al que pueden acceder fácilmente tanto los programadores en ensamblador, como los programadores en lenguaje C. Facilita el acceso a los periféricos internos y externos del 8051, haciendo innecesario la escritura en ensamblador de *drivers* para dispositivos periféricos.

Permite construir programas bien estructurados evitando los *goto* y los LJMP. De hecho, la mayor parte del código extra generado por C51 se emplea en asegurar una buena estructura para los programas, y no en un uso ineficiente del set de instrucciones del 8051.

Ofrece verdadera portabilidad desde el 8051 hacia otros procesadores, y a la inversa. Así las funciones existentes puede re-utilizarse, lo cual contribuye a reducir el tiempo de desarrollo de las aplicaciones.