ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba
EPSC

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA Y DE COMPUTADORES

SISTEMAS EMPOTRADOS

Apunte 2- LENGUAJE C PARA PROGRAMACIÓN A BAJO NIVEL

Profesor: Carlos Diego Moreno Moreno

Dpto. de Ingeniería Electrónica y de Computadores.
Área de Arquitectura y Tecnología de Computadores.
Escuela Politécnica Superior. Universidad de Córdoba.

UNIVERSIDAD DE CÓRDOBA

Objetivos.

- El presente documento tiene como objetivo profundizar en la programación en C para la programación a bajo nivel utilizada en los sistemas empotrados, ya que la mayoría de los sistemas empotrados tienen que interactuar con el *hardware* directamente.

Índice.

2.1.– Introducción.

- La mayoría de los sistemas empotrados han de interactuar directamente con el *hardware*. Para ello suele ser necesario manipular *bits* individuales dentro de los registros de configuración de los dispositivos.
- Por otro lado, este tipo de sistemas suelen ser también sistemas empotrados basados en microcontroladores, en los cuales, al no existir unidades de coma flotante, es necesario trabajar en coma fija. En estos casos es necesario conocer las limitaciones de los tipos de datos enteros: rango disponible, desbordamientos, etc.
- En este documento se van a estudiar precisamente estos dos aspectos, ya que éstos no suelen ser tratados en los libros de C de nivel introductorio o, si acaso, son tratados muy superficialmente.

Índice.

2.1.– Tipos de datos enteros.

- El lenguaje C dispone de 2 tipos de enteros básicos: `char` e `int`.
- El primero, aunque está pensado para almacenar un carácter, también puede ser usado para almacenar un entero de un *byte*, tanto con signo como sin signo (en este último caso se declarará la variable como `unsigned char`).
- Recuerde que, para el ordenador, un *byte* en la memoria es simplemente eso, un *byte*.
- Hasta que no se le dice al compilador que lo imprima como un carácter, dicho *byte* no se tratará como tal, realizando la traducción número–símbolo usando la tabla ASCII.

2.1.– Tipos de datos enteros.

- Otro aspecto a resaltar es la indefinición del tamaño de los tipos enteros en C.
- El lenguaje sólo especifica que el tipo `int` ha de ser del tamaño "natural" de la máquina. Así, en un microcontrolador de 16 *bits*, como los registros internos del procesador son de 16 *bits*, el tipo `int` tendrá un tamaño de 16 *bits*. En cambio, en un procesador de 32 *bits*, como el tamaño de los registros es de 32 *bits*, los `int` tienen un tamaño de 32 *bits*. Además, en un procesador de 8 *bits* podríamos encontrarnos con que un `int` tiene un tamaño de 8 *bits*, si el diseñador del compilador así lo ha decidido.
- Por si esto fuera poco, los tamaños de `short int` y de `long int` tampoco están determinados. Lo único a lo que obliga el lenguaje es a que el tamaño de un `short int` ha de ser menor o igual que el de un `int` y éste a su vez ha de tener un tamaño menor o igual que el de un `long int`.

2.1.– Tipos de datos enteros.

- Cuando se programa un sistema empotrado, una característica muy importante es la portabilidad. Se dice que un programa es portable cuando su código fuente está escrito de manera que sea fácil trasladarlo entre distintas arquitecturas, como por ejemplo pasar de un microcontrolador de 16 *bits* a otro de 32 *bits*.
- Por ejemplo, un programa escrito en C que use sólo funciones de la librería estándar, será fácilmente portable entre distintos tipos de ordenadores sin más que recompilarlo.
- Si se está escribiendo un programa de bajo nivel en el que se necesitan usar variables de un tamaño determinado, en lugar de usar los tipos básicos como `int` o `short int`, es mejor definir unos nuevos tipos mediante la sentencia `typedef`, de forma que si se cambia de máquina, baste con cambiar estas definiciones si los tamaños de los tipos no son iguales. Si no se hace así, habrá que recorrer todo el programa cambiando las definiciones de todas las variables, lo cual es muy tedioso y, además, propenso a errores.

2.1.– Tipos de datos enteros.

- Para definir los nuevos tipos, lo más cómodo es crear un archivo cabecera (al que se puede llamar por ejemplo **Tipos.h**) como el mostrado a continuación, e incluirlo en todos los programas.

```
#ifndef     TIPOS_H
#define     TIPOS_H
typedef     unsigned    char        uint8;
typedef     unsigned    short int   uint16;
typedef     unsigned    long int    uint32;
typedef     signed      char        int8;
typedef     signed      short int   int16;
typedef     signed      long int    int32;
#endif
```

2.1.– Tipos de datos enteros.

- El archivo **`Tipos.h`** no es más que una serie de definiciones de tipos en los que se menciona explícitamente el tamaño.
- Si se incluye este archivo en un programa, podrán usarse estos tipos de datos siempre que se necesite un tamaño de entero determinado.
- Por ejemplo, si se necesita usar un entero de 16 *bits* sin signo bastará con hacer:

```
#include "Tipos.h"
...
int main(void)
{
      uint16 mi_entero_de_16_bits_sin_signo;
      ...
}
```

Índice.

2.2.1.– Rangos de las variables enteras.

- Uno de los factores a tener en cuenta para elegir un tipo de dato entero es su rango.
- En general, el rango de un entero sin signo de n *bits* viene dado por la expresión: $R = 0 \rightarrow 2^n - 1$
- El rango de un entero con signo codificado en complemento a 2 es: $Rs = -2^{n-1} \rightarrow 2^{n-1} - 1$
- Por tanto, teniendo en cuenta estas ecuaciones, los rangos de los enteros de 8, 16 y 32 *bits* serán los mostrados en la tabla siguiente:

Bits	Rango sin signo	Rango con signo
8	0 ↔255	–128 ↔127
16	0 ↔ 65.535	–32.768 ↔32.767
32	0 ↔ 4.294.967.296	–2.147.483.648 ↔2.147.483.647

2.2.1.– Rangos de las variables enteras.

- Para elegir un tipo de entero u otro es necesario hacer un análisis del valor máximo que va a tomar la variable.
- Por ejemplo, si una variable va a almacenar la temperatura de una habitación, expresada en grados centígrados, será suficiente con una variable de 8 *bits* con signo (para tener en cuenta temperaturas bajo cero).
- Ahora bien, si es necesario realizar operaciones con esta temperatura, puede que sea necesario usar un tipo con mayor número de *bits*. Por ejemplo, en el siguiente fragmento de código:

```
int8 t;
t = LeeTemperaturaHabitacion();
t = t * 10;
```

2.2.1.– Rangos de las variables enteras.

- Si la temperatura de la habitación es por ejemplo 30 grados centígrados, el resultado de t * 10 será 300, que obviamente no cabe en una variable de 8 *bits*, produciéndose entonces un desbordamiento.
- El problema es que en C, para conseguir una mayor eficiencia, no se comprueba si se producen desbordamientos, con lo cual el error pasará desapercibido para el programa, pero no para el sistema que esté controlando.
- Es por tanto muy importante hacer una estimación de los valores máximos que puede tomar una variable, no solo en el momento de medirla, sino durante toda la ejecución del programa.

2.2.1.– Rangos de las variables enteras.

- De esta forma se podrá decidir qué tipo usar para la variable de forma que no se produzca nunca un desbordamiento.
- Por último, decir que en caso de usar un procesador de 32 *bits*, no se ahorra prácticamente nada en usar una variable de 8 o de 16 *bits*, por lo que, en vistas a una mayor seguridad, es mejor usar siempre variables de 32 *bits*.
- Si por el contrario el microprocesador es de 16 *bits*, en este caso sí que es mucho más costoso usar variables de 32 *bits*, por lo que sólo se deberán usar cuando sea necesario.

Índice.

2.3.– Conversiones de tipos.

- Las conversiones automáticas de C pueden ser peligrosas si no se sabe lo que se está haciendo cuando se mezclan tipos de datos.
- Normalmente las conversiones por defecto son razonables y funcionan en la mayoría de los casos.
- No obstante, pueden darse problemas como el mostrado a continuación:

```
int main()
{
      uint16 u;
      ...
      if (u > -1)   {
            printf ("Esto se imprime siempre.\n");
      }
}
```

2.3.– Conversiones de tipos.

- En este ejemplo se compara un entero sin signo (la variable u) con otro con signo (la constante –1).
- El lenguaje C, antes de operar dos datos de tipos distintos, convierte uno de ellos al tipo "superior" en donde por superior se entiende el tipo con más rango y precisión.
- Si se mezclan números con y sin signo, antes de operar se convierten todos a números sin signo.
- No obstante, la conversión no implica hacer nada con el patrón de *bits* almacenado en la memoria, sino sólo modificar cómo se interpreta.
- Así, en el ejemplo anterior la constante –1 queda convertida a 65535 (0xFFFF) que, según se puede ver en la tabla anterior, es el mayor número dentro del rango de `uint16`, por lo que la condición del `if` será siempre falsa.
- En conclusión, no se deben de mezclar tipos, salvo que se esté completamente seguro de lo que se está haciendo.

Índice.

2.3.1.– Categorías de conversiones.

- Las conversiones de tipos en C pueden clasificarse en dos categorías: **Promociones** y **degradaciones**.
- **PROMOCIONES**: Son aquellas conversiones en las que no hay pérdidas potenciales de precisión o rango. Un ejemplo es la conversión de un entero a un número en coma flotante:

```
double d = 4;
```

- Nótese que, en este ejemplo, sí que será necesario realizar un cambio en el patrón de *bits* que representa el número, ya que ambos se codifican de distinta manera. Esto obviamente conlleva una pequeña pérdida de tiempo.
- Si por el contrario se promociona un entero de 16 *bits* a otro de 32, sólo se añaden los *bits* más significativos. Si el número es sin signo, se añaden ceros a la izquierda hasta completar el tamaño de la nueva palabra. Si por el contrario el número es con signo, se extiende el *bit* de signo para no modificar el valor del número. La extensión de signo consiste en copiar en los *bits* más significativos que se añaden el mismo valor que tiene el *bit* de signo.

2.3.1.– Categorías de conversiones.

- **DEGRADACIONES**: Son aquellas en las que se produce una pérdida de precisión, como por ejemplo la asignación de un número en coma flotante a un entero (`int i = 4.3;`). En este caso se pierde la parte decimal. Otro ejemplo de degradación es la asignación de un número de 32 *bits* a uno de 16:

```
int32 i_largo;
int16 i_corto;
...
i_corto = i_largo; /* Posible error */
```

- En este caso, se pierden los 16 *bits* más significativos, por lo que si el valor almacenado en `i_largo` es mayor que 32767 o menor que –32768, se producirá un error al salirse el valor del rango admisible por un `int16`.

2.3.1.– Categorías de conversiones.

- Por último, conviene recordar que es posible forzar una conversión mediante el operador `cast`. El operador `cast` consiste en el tipo al que se desea convertir el operando encerrado entre paréntesis.
- Este operador es necesario, por ejemplo, al asignar memoria dinámica para convertir el puntero genérico (`void *`), devuelto por la función `calloc`, al tipo de la variable a la que se le asigna memoria:

```
double *pd;
...
pd = (double *)calloc(20, sizeof(double));
```

2.3.1.– Categorías de conversiones.

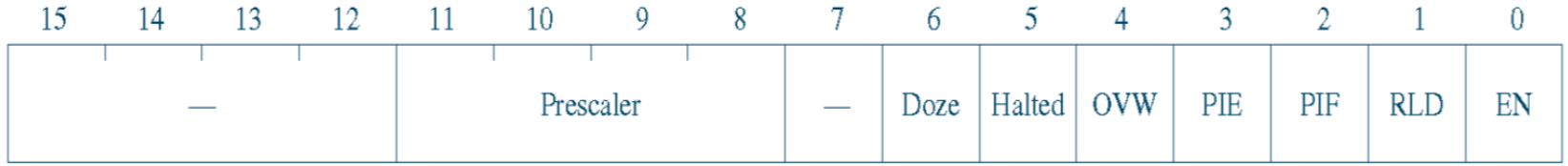
- También es necesario el uso del operador `cast` al realizar una degradación si se desea evitar que el compilador genere un aviso, pues de esta forma se le dice al compilador que se está seguro de lo que se está haciendo (o al menos eso se espera):

```
int32 i_largo;
int16 i_corto;
...
i_corto = (int16) i_largo;   /* Estamos seguros que i_largo contiene un valor */
                             /* que está dentro del rango de int16 */
```

Índice.

2.4.– Manipulación de bits.

- En la programación de sistemas empotrados es muy frecuente encontrarse con variables que, en lugar de un número, contienen una serie de campos de *bit* en los que se agrupa información diversa.
- Un ejemplo típico son los registros de configuración de los periféricos. Por ejemplo, en la figura se muestra el registro de configuración de los temporizadores PIT (*Programmable Interrupt Timer*) del *ColdFire MCF5282*. Este registro de 16 *bits* contiene un campo de 4 *bits* (*Prescaler*) y 7 campos de 1 *bit* (*Doze*, *Halted*, OVW, PIE, PIF, RLD y EN). Obviamente, para manejar los temporizadores es necesario poder acceder a cada uno de estos campos por separado.



2.4.– Manipulación de bits.

- Otro ejemplo típico consiste en empaquetar varios datos en una sola variable, lo cual sólo tiene sentido si la memoria del microcontrolador es escasa; ya que el acceso a cada una de las variables empaquetadas es más complejo.
- Por ejemplo, si tenemos 8 variables lógicas, en lugar de usar 8 *bytes* para almacenarlas, podemos usar un sólo *byte* y asignar un *bit* para cada variable lógica.
- Dado que el lenguaje C se pensó para realizar programas de bajo nivel, es decir, programas que interactuasen directamente con el *hardware*, se definieron varios métodos para manipular variables a nivel de *bit*; los cuales se estudian en las siguientes secciones.

Índice.

2.4.1.– Operadores a nivel de *bit*.

- El lenguaje C define los siguientes operadores a nivel de *bit*:
  - **Operadores de desplazamiento**. Los operadores **<<** y **>>** permiten desplazar el valor de una variable un número de *bits* hacia la izquierda o la derecha respectivamente. Por ejemplo:

```
b = a << 4;
```

desplaza el valor almacenado en **a** 4 *bits* hacia la izquierda y almacena el resultado en **b**. Recuerde que desde el punto de vista aritmético, desplazar **n** *bits* a la izquierda equivale a multiplicar por 2^n. De la misma forma desplazar **n** *bits* a la derecha equivale a dividir por 2^n.

2.4.1.– Operadores a nivel de *bit*.

- **Operadores lógicos a nivel de *bit* de dos operandos.** Los operadores **&, |** y **^** realizan las operaciones **AND**, **OR** y **XOR** *bit* a *bit* entre sus dos operandos. Así, si se inicializan a y b con los valores:

```
a = 01101101
b = 10101001
```

  Entonces el resultado de hacer `c = a & b;` será `00101001`

- **El operador ~ (NOT)** invierte cada uno de los *bits* de su operando. Por ejemplo si `a=0xF0`, `~a` valdrá `0x0F`.

- **NOTA** : los operadores lógicos y los operadores lógicos a nivel de *bit* no son equivalentes.

2.4.1.– Operadores a nivel de *bit*.

- **Manipulación de *bits* individuales:** Los operadores de desplazamiento y de nivel de *bit* son útiles para manipular *bits* individuales dentro de una palabra. Los casos típicos son:
  - **Verificar el estado de un bit de una variable.** Para ello basta con hacer un AND entre la variable y una máscara con todos los *bits* a cero excepto el *bit* que se quiere comprobar. Así, para verificar el estado del *bit* PIF (bit 2) del registro PCSR del temporizador PIT0 del microcontrolador MCF5282 (MCF_PIT0_PCSR), hay que escribir:

```
if (MCF_PIT0_PCSR & (1<<2)) /* ¿bit 2 a 1? */
```

Como se puede apreciar, aunque podría haberse calculado el valor de la máscara (0x0004), es mucho más fácil utilizar el operador desplazamiento, ya que así se aprecia directamente el número del *bit* a verificar. Además, la operación de desplazamiento se evalúa en tiempo de compilación ya que sus dos argumentos son constantes. En consecuencia, el programa final se ejecutará igual de rápido.

2.4.1.– Operadores a nivel de *bit*.

- **Para poner un *bit* a 1** hay que hacer una OR entre la variable y una máscara con todos los *bits* a cero, excepto el *bit* que se quiere poner a 1. El método para obtener la máscara es idéntico al caso anterior. Así, el siguiente código pone a uno el *bit* PIE (bit 3) del registro PCSR del temporizador PIT0:

```
MCF_PIT0_PCSR |= (1<<3); /* bit 3 a 1 */
```

- **Para poner un *bit* a 0,** la máscara ha de construirse con todos los *bits* a 1 excepto el *bit* n y hacer un AND entre la variable y la máscara. Para ello, primero se crea una máscara con todos los *bits* a cero excepto el *bit* n y luego se invierte la máscara con el operador NOT. Por ejemplo, si se desea poner a cero el *bit* PIE (*bit* 3) del registro PCSR del temporizador PIT0, basta con hacer:

```
MCF_PIT0_PCSR &= ~(1<<3); /* bit 3 a 0 */
```

- **Para invertir un *bit*** se hace una XOR con una máscara igual a la usada para ponerlo a 1:

```
MCF_PIT0_PCSR ^= (1<<3); /* bit 3 se invierte */
```

2.4.1.– Operadores a nivel de *bit*.

- **Manipulación de campos de *bits*:** La manipulación de campos de varios *bits* dentro de una misma palabra es parecida a la manipulación de *bits* sueltos. Los casos típicos son:

  - **Extraer un campo de *bits* para analizarlo.** Para ello, los pasos a seguir son:
    - Desplazar el dato a la derecha para llevar el campo al *bit* 0.
    - Hacer una AND con una máscara para eliminar el resto de *bits*.

    Por ejemplo, si se desea obtener el valor del campo *Prescaler* del registro PCSR del temporizador PIT0, habrá que desplazar el dato 8 *bits* para trasladar el campo del *bit* 8 al bit 0. A continuación, habrá que hacer una AND con la máscara 0x000F, para poner los *bits* 15 a 4 a cero y dejar así sólo el campo *Prescaler*. La instrucción para llevar esto a cabo es:

```
pres = (MCF_PIT0_PCSR >>8) & 0x000F;
```

2.4.1.– Operadores a nivel de *bit*.

- **Escribir un campo de *bits*.** En este caso, el proceso es:
  - Borrar el campo de *bits*.
  - Eliminar los *bits* más significativos que sobren del dato a escribir. Si se está seguro de que el valor a escribir tiene los *bits* que no forman parte del campo a cero, este paso puede ser innecesario. Un ejemplo típico es la escritura de una constante. Si no se está seguro, es mejor aplicar la máscara para una mayor robustez del programa
  - Desplazar el dato a escribir para alinearlo con el campo destino.
  - Hacer una OR entre el dato a escribir y la variable destino.

  Por ejemplo, para escribir el valor de la variable `pres` en el campo *Prescaler* del registro PCSR del temporizador PIT0, hay que ejecutar las siguientes instrucciones:

```
MCF_PIT0_PCSR &= ~(0x0F << 8); /* Puesta a cero */
MCF_PIT0_PCSR |= (pres & 0x000F) << 8
```

2.4.1.– Operadores a nivel de *bit*.

Uso de máscaras:

- Para conseguir una mayor claridad del código, pueden definirse máscaras para acceder a *bits* individuales, y darles a estas máscaras los nombres de los *bits*. Por ejemplo, para poder modificar el estado de los *bits* PIE y PIF se definen las máscaras:

```
#define PIF (1<<2)
#define PIE (1<<3)
```

- Entonces, para poner a 1 el *bit* PIE del registro MCF_PIT0_PCSR se hará un OR con su máscara:

```
MCF_PIT0_PCSR |= PIE;
```

- Y para ponerlo a 0, se hará una AND con su máscara negada:

```
MCF_PIT0_PCSR &= ~PIE;
```

2.4.1.– Operadores a nivel de *bit*.

Uso de máscaras:

- La ventaja de este método es que las definiciones pueden situarse en un archivo cabecera (por ejemplo `mcf5282.h`). A partir de entonces, no se tendrá que volver a mirar el manual para averiguar en qué posición está el *bit*: sólo habrá que recordar el nombre del *bit*, lo cual es siempre muchísimo más fácil.
- Si se desean activar o desactivar varios *bits* a la vez, pueden combinarse varias máscaras con el operador `|`. Por ejemplo para poner a 1 los *bits* PIE y PIF:

```
MCF_PIT0_PCSR |= PIE | PIF;
```

- Y para ponerlo a 0:

```
MCF_PIT0_PCSR &= ~(PIE | PIF);
```

- Por último, conviene tener en cuenta que algunos microcontroladores, disponen de direccionamientos a nivel de *bit* que permiten acceder a *bits* individuales sin necesidad de usar máscaras. No obstante, si se busca un programa portable es mejor usar máscaras y operadores a nivel de *bit*, ya que éstos están soportados por todas las arquitecturas y compiladores.

Índice.

2.4.2.– Campos de *bits*.

- A pesar de usar nombres para las máscaras, el acceso a campos de *bits* dentro de una palabra usando operadores lógicos y desplazamientos no es muy intuitivo.
- Para facilitar la escritura del código y su legibilidad, el lenguaje C incluye un método para acceder a estos campos de *bits* usando una sintaxis similar a la de las estructuras.
- La única diferencia entre una estructura normal y una estructura con campos de *bits* radica en que cada variable se puede dividir en una serie de campos de *bits* de tamaños arbitrarios, siempre y cuando estos tamaños sean inferiores al del dato sobre el que se declaran.
- En el ejemplo siguiente se define un tipo compuesto por una palabra de 8 *bits* dividida en dos campos de 4 *bits* para almacenar un número BCD de dos dígitos.

2.4.2.– Campos de *bits*.

```
typedef struct{
      uint8 digito0 :4, /* Ojo se termina con una coma
            digito1 :4; /* Ojo el final es un ; */
}BCD2;
```

- Nótese que los campos se separan con una coma, mientras que el final de la variable se indica con un punto y coma.
- Esta sintaxis es así para poder incluir varias variables con campos de bits dentro de una estructura, como por ejemplo:

```
typedef struct{
    uint8 digito0 :4,
          digito1 :4;
    uint8 digito2 :4,
          digito3 :4;
}BCD4;
```

2.4.2.– Campos de *bits*.

- El acceso a los elementos de una estructura de campos de *bits* se realiza del mismo modo que el acceso a elementos de estructuras normales: usando el operador punto o el operador flecha (->) si se dispone de un puntero a la estructura.
- En el ejemplo siguiente, se muestra un programa que define un número BCD de dos dígitos y que a continuación lo inicializa al valor 27.

```
int main (void)
{
    BCD2 numero;
    numero.digito0 = 2;
    numero.digito1 = 7;
    ...
}
```

2.4.2.– Campos de *bits*.

- Es necesario volver a recalcar que la sintaxis de las estructuras con campos de *bits* es exactamente la misma que la de las estructuras normales.
- Así, una estructura de campos de *bits* también puede inicializarse al declararla. Por ejemplo, el código anterior también puede escribirse como:

```
int main (void)
{
    BCD2 numero = {2, 7};
    ...
}
```

- En este caso, el primer campo definido en la estructura (digito0) se inicializa con el primer valor (2) y el segundo campo (digito1) con el segundo valor (7).

2.4.2.– Campos de *bits*.

- También pueden crearse vectores de estructuras de campos de *bits* e inicializar dichos vectores:

```
int main (void)
{
     BCD2 numeros[] = {2, 7, 4, 0};
     ...
}
```

- En este caso el primer elemento del vector se inicializará con el número BCD 27 y el segundo con el 40.
- Además de para trabajar con datos definidos por el programador, como en el ejemplo anterior de números BCD, también se puede usar una estructura con campos de *bit* para acceder a los registros de configuración de los periféricos de un microcontrolador.

2.4.2.– Campos de *bits*.

- Para ello sólo hay que definir adecuadamente los campos. El lenguaje C permite incluso definir campos vacíos para tener en cuenta los *bits* no usados dentro del registro.
- Así, en el siguiente ejemplo se muestra la declaración de una estructura de campos de *bits* para acceder cómodamente al registro MCF_PIT0_PCSR.

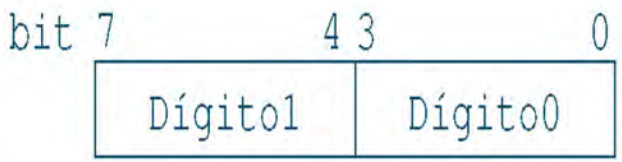
```
typedef struct{
  uint16 :4, /* No usado */
         Prescaler    :4,
                      :1, /* No usado */
         Doze :1,
         Halted :1,
         OVW :1,
         PIE :1,
         PIF :1,
         RLD :1,
         EN :1; /* Ojo es ; */
}MCF_PIT0_PCSR_campos; /* Campos de bits */
```

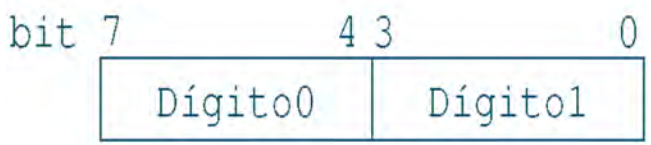
2.4.2.– Campos de *bits*. Inconvenientes del manejo de campos de *bits* en C.

- Aunque el uso de campos de *bits* mejora la legibilidad del código, presenta dos inconvenientes importantes, los cuales hacen que en la práctica su uso no esté muy extendido.
- El primer inconveniente es la eficiencia, pues para acceder a cada campo es necesario realizar las operaciones de desplazamiento y enmascaramiento expuestas en la sección 2.4.1.
- Esto impide realizar optimizaciones como poner a uno o a cero varios *bits* a la vez.
- También es más eficiente acceder a una variable estándar (`int`, `char`, etc.) que a un campo de *bits*.
- Por tanto, su uso para almacenar variables, tal como se ha mostrado en los ejemplos anteriores que trabajan con números en BCD, sólo estará justificado si la memoria del ordenador está limitada.

2.4.2.– Campos de *bits*. Inconvenientes del manejo de campos de *bits* en C.

- El segundo inconveniente, más grave si cabe, es que el orden en el que se colocan los campos dentro de la palabra no está definido por el lenguaje.
- Así, en los ejemplos anteriores con los números BCD, un compilador puede colocar los campos declarados en primer lugar en los *bits* menos significativos y otro los puede colocar en los más significativos.
- Por ejemplo, el compilador `gcc` para IA–32 en *Linux* coloca en los *bits* menos significativos los primeros campos en definirse (*Little Endian*). En cambio, el compilador `CodeWarrior` para *ColdFire* coloca el primer campo de la variable en los *bits* más significativos (*Big Endian*).





2.4.2.– Campos de *bits*. Inconvenientes del manejo de campos de *bits* en C.

- La razón de este desaguisado radica en que el estándar de C no obliga a ningún orden en especial. Por tanto, los diseñadores del compilador hacen que el orden de los campos de *bits*, siga al de los *bytes* dentro de una palabra.
- Los procesadores de la familia IA–32 almacenan el *byte* menos significativo de la palabra, en la posición baja de la zona de memoria en la que se almacena dicha palabra. A este tipo de arquitectura se le denomina ***Little Endian***. En este tipo de arquitecturas, el compilador coloca los primeros campos en definirse en los *bits* menos significativos de la palabra.
- En cambio, el *ColdFire* es una arquitectura ***Big Endian***, lo cual quiere decir que el *byte* más significativo se coloca en la posición de memoria más baja. Por ello, en esta arquitectura el compilador coloca los primeros campos en definirse, en los *bits* más significativos de la palabra.

2.4.2.– Campos de *bits*. Inconvenientes del manejo de campos de *bits* en C.

- Esto no tiene importancia si el uso de los campos es sólo para conseguir un almacenamiento de los datos más compacto, tal como se ha hecho con los números BCD.
- Sin embargo, si se definen campos de *bits* para acceder a registros internos del microcontrolador, tal como el ejemplo mostrado para acceder al registro MCF_PIT0_PCSR, el orden de los campos es obviamente muy importante.

Índice.

2.5.– Acceso a registros de configuración del microcontrolador.

- Los periféricos que incluyen los microcontroladores se configuran y se controlan por medio de una serie de registros.
- Estos registros están situados en posiciones fijas de memoria definidas por el diseñador del *hardware*.
- Cuando se define una variable en C, el compilador asigna un espacio de memoria para almacenarla y luego usa su dirección internamente para acceder a la variable y usar su contenido.
- Sin embargo, en el caso de los registros de configuración, el espacio de memoria ya está asignado, por lo que sólo es necesario definir un puntero a la posición de memoria del registro.

2.5.– Acceso a registros de configuración del microcontrolador.

- Por ejemplo, si se necesita acceder en un programa al registro PCSR del microcontrolador *MCF5282*, en primer lugar habrá que consultar el manual del microcontrolador para saber en qué posición de memoria está dicho registro.
- Si buscamos en el manual del microcontrolador *FreeScale*, dicho registro está en la dirección `0x40150000.15` A continuación se muestra un ejemplo para esperar el final de la cuenta del temporizador, comprobando para ello el *bit* PIF (*bit* 2) del registro. Dicho *bit* se pone a 1 al finalizar la cuenta del temporizador.

```
#define PIF (1<<2)
...
void EsperaFinTemp(void)
{
        volatile uint16 *p_pcsr = (volatile uint16 *)0x40150000;
        while( (*p_pcsr & PIF) == 0); /* Espera fin temporizador */
}
```

2.5.– Acceso a registros de configuración del microcontrolador.

- En primer lugar conviene destacar que la variable `p_pcsr` es un puntero a una variable de tipo `uint16`, ya que el registro PCSR es de 16 *bits*.
- Dicho puntero se ha inicializado a la dirección del registro, con lo cual, usando el operador `*` podremos acceder al contenido del registro, tal como se muestra en la línea del `while`.
- Nótese que la dirección del registro es una constante entera, por lo que es necesario usar un `cast` para convertirla a un puntero y evitar así un aviso (*warning*) del compilador.
- Por otro lado, hay que aclarar el porqué se ha usado la palabra clave `volatile` en la declaración de la variable.

2.5.– Acceso a registros de configuración del microcontrolador.

- Si se declara la variable `p_pcsr` como una variable normal, el compilador para optimizar el código cargará la variable en un registro antes de entrar en el bucle y luego, en lugar de volver a leer la variable de memoria en cada iteración, usará la copia del registro, que es mucho más eficiente.
- El problema radica en que cuando el temporizador termine y cambie el *bit* 2 del registro MCF5282_PIT0_PCSR, el programa no se enterará, puesto que estará comprobando el valor antiguo que guardó en el registro.
- El programa por tanto se quedará en un bucle infinito. Para evitar esto, mediante la palabra clave `volatile` se informa al compilador que la variable puede cambiar por causas externas a la ejecución del programa.
- El compilador entonces se verá obligado a leer siempre la palabra de la memoria, en lugar de usar una copia guardada en un registro interno.

Índice.

2.5.1.– Acceso a registros internos sin usar una variable de tipo puntero.

- Los pasos seguidos en el ejemplo anterior para acceder a un registro de configuración, pueden realizarse sin necesidad de usar una variable auxiliar de tipo puntero para almacenar la dirección del registro.
- La misma función puede escribirse así:

```
#define PIF (1<<2)
...
void EsperaFinTemp(void)
{
     while( (*(volatile uint16 *)0x40150000 & PIF) == 0); /* Espera fin temporizador */
}
```

2.5.1.– Acceso a registros internos sin usar una variable de tipo puntero.

- No obstante, en estos casos es mejor definir una constante para mejorar la legibilidad del código:

```
#define PIF (1<<2)
#define PCSR_PIT0 (*(volatile uint16 *)0x40150000)
...
void EsperaFinTemp(void)
{
     while( (PCSR_PIT0 & PIF) == 0); /* Espera fin temporizador */
}
```

- Esta es la alternativa usada en el entorno de desarrollo *CodeWarrior* para acceder a los registros del microcontrolador.

Índice.

2.6.– Uniones.

- Una unión es similar a una estructura salvo que, en lugar de reservarse una zona de memoria para cada miembro de la estructura, se reserva una sola zona de memoria a compartir por todos los miembros de la unión.
- En el ejemplo siguiente, se reserva una sola palabra de 16 *bits*.

```
typedef struct{
        uint8 byte1;
        uint8 byte2;
}DOSBYTES;
typedef union{
        uint16 palabra;
        DOSBYTES bytes;
}U_WORD_BYTE;
```

2.6.– Uniones.

- A esta zona de memoria se puede acceder o bien como una palabra de 16 *bits* o bien como una estructura formada por dos *bytes*. Esto permite escribir en la unión una palabra para luego poder acceder a los dos *bytes* que la componen, lo cual es útil si se desea invertir el orden de ambos *bytes*.
- Por ejemplo, la siguiente función invierte el orden de los dos *bytes* de una palabra, lo cual es necesario si se desea enviar una palabra desde una máquina *Little Endian* a una máquina *Big Endian*.

```
uint16 swap(uint16 ent)
{
        U_WORD_BYTE uwb;
        uint8 temp;
        uwb.palabra = ent;
        temp = uwb.bytes.byte1;
        uwb.bytes.byte1 = uwb.bytes.byte2;
        uwb.bytes.byte2 = temp;
        return uwb.palabra;
}
```

Índice.

2.7.– Extensiones del lenguaje.

- Los compiladores ofrecen ciertas extensiones al lenguaje que permiten, entre otras cosas, un acceso al *hardware* que no se previó en el estándar.
- En esta sección se van a discutir dos extensiones que son imprescindibles en los programas de bajo nivel: escritura de instrucciones en ensamblador y soporte de interrupciones.
- El problema de estas extensiones es que no son estándares, por lo que cada compilador implanta las que sus diseñadores creen más convenientes.
- No obstante, todos los compiladores suelen tener las extensiones que se van a discutir aquí, aunque con distinta sintaxis.

Índice.

2.7.1.– Uso de ensamblador en C.

- Cuando se realizan programas de bajo nivel, hay situaciones en las que no hay más remedio que usar instrucciones en código máquina.
- Por ejemplo, en el *ColdFire*, para habilitar las interrupciones hay que usar una instrucción de código máquina especial que copia un valor en el registro de estado de la CPU (denominado SR).
- Para facilitarle la vida al programador, la mayoría de los compiladores disponen de mecanismos que permiten introducir instrucciones en ensamblador dentro de una función en C.

2.7.1.– Uso de ensamblador en C.

- En el caso de *CodeWarrior*, basta con usar la directiva `asm`, encerrando entre llaves el código en ensamblador; tal como se muestra en el siguiente ejemplo:

```
void EnableInt(void)
{
        asm{
        move.w #0x2000 ,SR
        }
}
```

2.7.1.– Uso de ensamblador en C.

- Incluso es posible acceder a las variables definidas en la función desde el código en ensamblador, tal como se muestra en el siguiente ejemplo para *CodeWarrior*:

```
long square(short a)
{
        asm {
                move.w a,d0 // Copia la variable a al registro
                mulu.w d0,d0 // lo eleva al cuadrado
        }
        return;
/* Por convención las funciones devuelven el resultado en el
registro D0. Como ya se ha puesto el resultado en D0 en el
ensamblador , no hace falta poner nada en el return. */
}
```

2.7.1.– Uso de ensamblador en C.

- Conviene destacar que en estos casos el compilador se encarga de generar el código para usar los argumentos, crear variables locales, salir de la función, etc.
- Es por ello que en lugar de usar la instrucción de código máquina `rts` para salir de la función desde el código en ensamblador, se ha usado la instrucción de C `return` que hace que el compilador genere el código necesario para salir de la función ordenadamente.

Índice.

2.7.2.– Soporte de interrupciones.

- El estándar ANSI C 1999 no contempla el soporte de interrupciones, ya que éstas dependen del procesador que se esté usando.
- Por ejemplo, hay procesadores como el MIPS que cuando se produce una interrupción saltan siempre a una posición de memoria determinada, mientras que otros como el *ColdFire* o el *Infineon* 167, saltan a una posición de memoria en función de la interrupción producida.
- En estos casos se dice que el procesador tiene un sistema de interrupciones vectorizadas, ya que se construye en la memoria un vector de direcciones de forma que cuando se produce la interrupción número n se salta a la posición n del vector.
- Por ejemplo, el 167 salta a la posición 0x20 del vector de interrupciones cuando se produce una interrupción del temporizador 0.

2.7.2.– Soporte de interrupciones.

- También conviene tener en cuenta que una rutina de atención a interrupción es distinta de una función normal.
- Para empezar, como puede ser llamada en cualquier instante, ha de guardar cualquier registro que vaya a modificar.
- Además, la instrucción de código máquina para retornar de la rutina de interrupción es distinta de la de una función normal (en el salto a una función normal sólo se guarda el contador de programa, sin embargo, cuando se salta a una rutina de interrupción ha de guardarse además el registro de estado como mínimo).
- Por último, conviene recordar que las rutinas de atención a interrupción no devuelven ni reciben ningún valor.

2.7.2.– Soporte de interrupciones.

- Afortunadamente, la mayoría de los compiladores disponen de extensiones para el soporte de interrupciones.
- Así, en el compilador de *Keil* para *Infineon 167*, la definición de una rutina de atención a interrupción se realiza añadiendo la palabra clave `interrupt` seguida del número del vector de interrupción al que se asociará la función, tal como se muestra a continuación:

```
void InterruptTimer0(void) interrupt 0x20
{
        /* Rutina de atención a la interrupción del timer 0 */
        /* Compilador Keil para 167 */
}
```

2.7.2.– Soporte de interrupciones.

- En cambio, el compilador *CodeWarrior* para *ColdFire* sólo permite definir una función como de atención a interrupción.
- Para ello se precede su definición con la directiva `__declspec(interrupt)`, tal como se muestra en el siguiente ejemplo.
- Sin embargo, al contrario que el compilador *Keil*, deja al programador la labor de inicializar el vector de interrupción correspondiente, para que apunte a la rutina de atención a interrupción:

```
__declspec(interrupt) void InterruptPIT0(void)
{
        /* Rutina de atención a la interrupción de PIT0 */
        /* Compilador CodeWarrior para ColdFire */
}
```

2.7.2.– Soporte de interrupciones.

- Por último, a continuación se muestra otro ejemplo de declaración de rutina de atención a interrupción, en este caso para el compilador de software libre *gcc*:

```
void __attribute__((interrupt("IRQ")))
        SYS_kbd_irq_handler(void)
{
        /* Rutina de atención a la interrupción del teclado */
        /* Compilador gcc para IA-32 (Linux) */
}
```

- Este compilador, al igual que *CodeWarrior*, deja al programador la labor de inicializar el vector de interrupción.

2.7.2.– Soporte de interrupciones.

- El que *CodeWarrior* y *gcc* no inicialicen el vector de interrupción es debido a que estos compiladores están orientados a microprocesadores más complejos que permiten situar el origen de los vectores de interrupción en una posición arbitraria de memoria, por lo que el compilador no puede saber a priori dónde ha de colocar la dirección de la rutina de interrupción.
- Por el contrario, en el 167 la tabla de vectores de interrupción está siempre en el mismo sitio, con lo que el compilador puede encargarse de esta tarea.
- NOTA: En ambos casos: `__declspec(interrupt)` y `__attribute__` las palabras clave están precedidas por dos guiones bajos.

Bibliografía.

- *Sistemas Empotrados en Tiempo Real*. Capítulo 2. Una introducción basada en FreeRTOS y en el microcontrolador ColdFire MCF5282 José Daniel Muñoz Frías. Octubre 2008.

¡Muchas gracias por su atención!

Carlos Diego Moreno Moreno

Área de Arquitectura y Tecnología de Computadores
Departamento de Ingeniería Electrónica y Computadores.
Escuela Politécnica Superior. Universidad de Córdoba