SASE 2013

Programación en C para Sistemas Embebidos

(con ejemplos basados en MSP430)

Mg. Guillermo Friedrich
UTN-FRBB

Tópicos destacados

- Generalidades sobre la arquitectura MSP430
- Paso de parámetros a funciones y variables locales
- Uso de estructuras y punteros a estructuras
- Estructuras y uniones “anónimas”
- Campos de bits
- Constantes
- Variables asociadas a SFR del micro
- Proteger la concurrencia sobre variables compartidas
- Atención de interrupciones
- Fijar direcciones de memoria usadas por código y datos

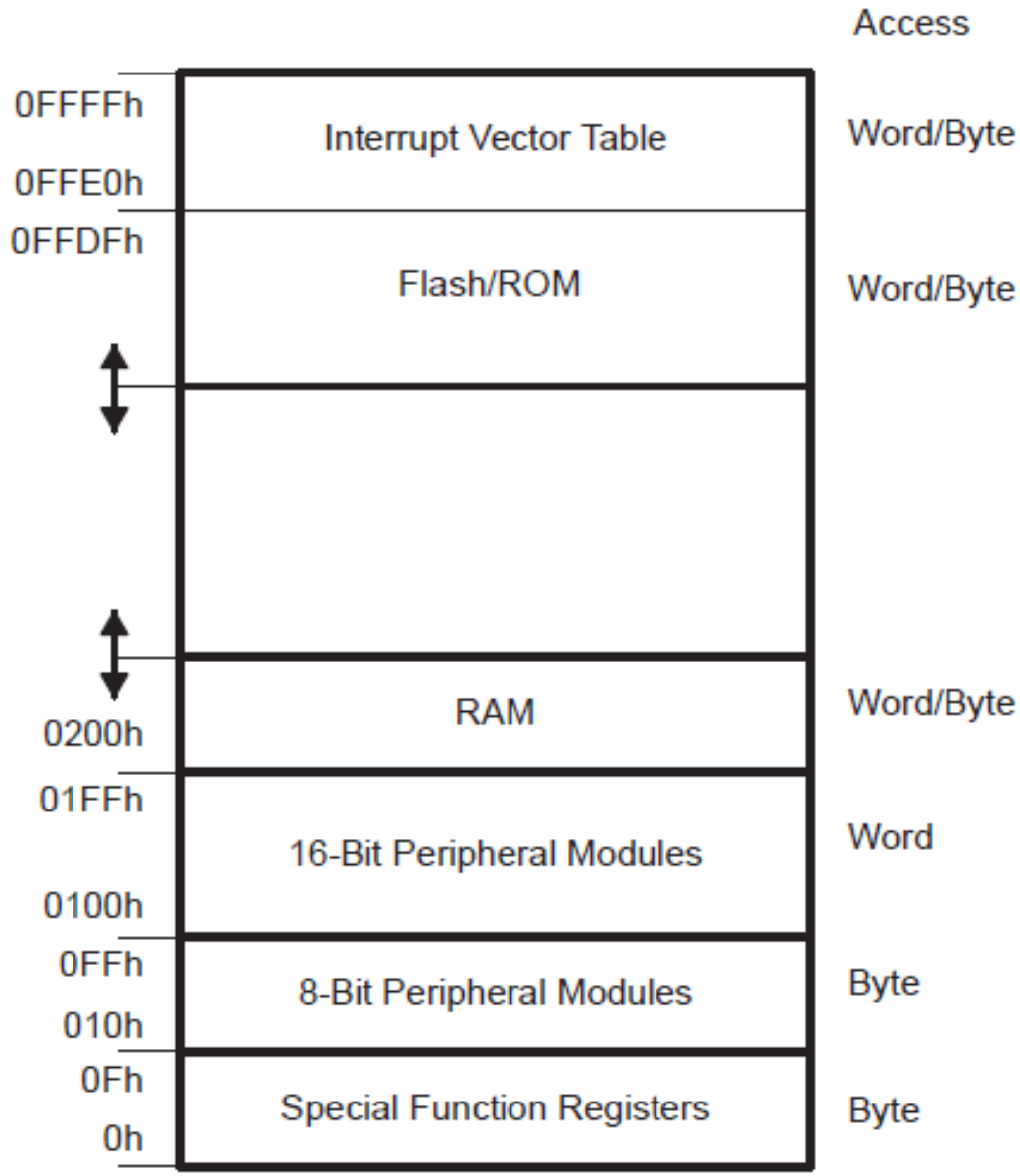
MSP430

- Arquitectura RISC 16-Bit, Ciclo de Instrucción 62.5 o 125-ns

- MSP430F133: 8KB+256B Flash Memory, 256B RAM
- MSP430F135: 16KB+256B Flash Memory, 512B RAM
- MSP430F147: 32KB+256B Flash Memory, 1KB RAM
- MSP430F148: 48KB+256B Flash Memory, 2KB RAM
- MSP430F149: 60KB+256B Flash Memory, 2KB RAM

- MSP430G2553: 16KB+256B Flash Memory, 512 B RAM
- MSP430G2211: 2KB+256B Flash Memory, 128 B RAM

MSP430: mapa de memoria



- La reducida cantidad de RAM obliga a tener en cuenta algunos detalles al programar en C.

- Los parámetros y variables locales (auto) de las funciones utilizan registros del procesador y a veces también la pila.

- Las variables con valores constantes (por ej. tablas de parámetros, textos predefinidos, etc.) conviene declararlas como ***const***, para que sean ubicadas en memoria Flash.

Codificación eficiente para aplicaciones embebidas

- Seleccionar los tipos de datos más adecuado.

- Controlar la ubicación de los objetos de código y datos en memoria.

- Controlar las optimizaciones del compilador.

- Escribir código eficiente.

Selección de los tipos de datos

- Usar los tipos más pequeños que sea posible.

- Tratar de evitar el uso de *double* y *long long* (64 bits)

- No usar campos de bit de tamaño mayor que 1.

- Tratar de evitar el uso de punto flotante.

- Si un puntero no se va a usar para modificar datos, declararlo *const* (Por ej.: *const char *p*).

Uso de punto flotante

- En un micro sin coprocesador matemático, las operaciones de punto flotante son ineficientes en tiempo de ejecución y uso de memoria.

- Si se puede, es preferible usar tipos ***float*** (32 bits) en lugar de ***double*** (64 bits).

- Las constantes de punto flotante por defecto son ***double***, excepto que se las especifique como ***float***.

```
a += 1.0 // double      a += 1.0f  // float
```

Campos de bits

- Por ejemplo, dada la siguiente estructura de campos de bits:

```
union bits
{
  struct
  {
    unsigned int b1:1;
    unsigned int b2:2;
    unsigned int b3:3;
    unsigned int b4:4;
    unsigned int b6:6;
  }b;

  unsigned int w;
}bf;
```

Campos de bits

- Las operaciones sobre campos de un bit pueden requerir una sola instrucción de máquina.

En C:

```
bf.b.b1 = 1;
```

En lenguaje ensamblador:

```
bis.w   #0x1,&bf
```

Campos de bits

- Las operaciones sobre campos de más de un bit requieren varias instrucciones de máquina:

En C:

```
bf.b.b3 = 5;
```

En lenguaje ensamblador:

```
mov.w   &bf,R15        ; copia
and.w   #0xFFC7,R15    ; borra
bis.w   #0x28,R15      ; setea
mov.w   R15,&bf        ; copia
```

Campos de bits

- La misma operación implementada sin campos de bits es más eficiente:

En C:

```
w &= 0x00c7;  // borra 3 bits
w |= 5 << 3;  // escribe el 5
```

En lenguaje ensamblador:

```
and.w   #0xC7,&w  ; borra
bis.w   #0x28,&w  ; escribe
```

Estructuras y Uniones anónimas

- Son propias del lenguaje C++

- Algunos compiladores de C tienen extensiones que las admiten.

- Los nombres de los miembros tienen alcance fuera de la estructura o unión.

Estructuras y Uniones anónimas

```
union bits
{
  struct
  {
    unsigned int b1:1;
    unsigned int b2:2;
    unsigned int b3:3;
    unsigned int b4:4;
    unsigned int b6:6;
  }b;

  unsigned int w;
}bf;
```

```
// unnamed
union
{
  struct
  {
    unsigned int b1:1;
    unsigned int b2:2;
    unsigned int b3:3;
    unsigned int b4:4;
    unsigned int b6:6;
  };

  unsigned int w;
};
```

Estructuras y Uniones anónimas

```
// unnamed
union
{
  struct
  {
    unsigned int b1:1;
    unsigned int b2:2;
    unsigned int b3:3;
    unsigned int b4:4;
    unsigned int b6:6;
  };

  unsigned int w;
};
```

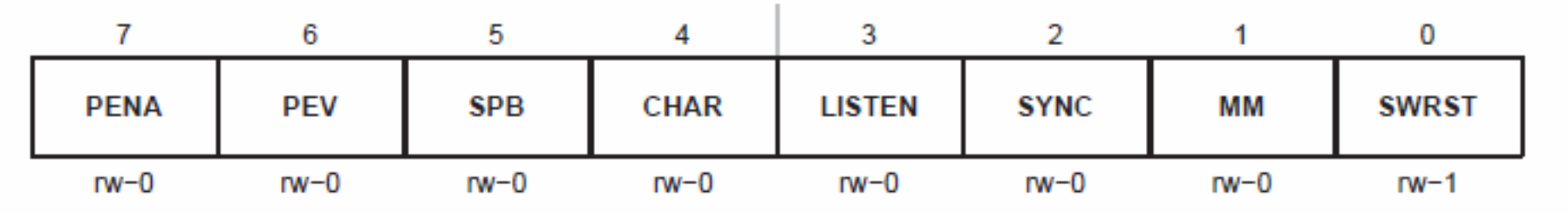
```
// los miembros de las
// struct y union
// unnamed tienen
// alcance global,
// no pueden repetirse

main()
{  // Pone todo a 0
   w = 0;
   // asigna valores
   // a algunos campos
   b3 = 5;
   b6 = 9;
}
```

Estructuras y Uniones anónimas

- Las estructuras y uniones anónimas son de utilidad para manipular los registros del procesador que están mapeados en direcciones de memoria.

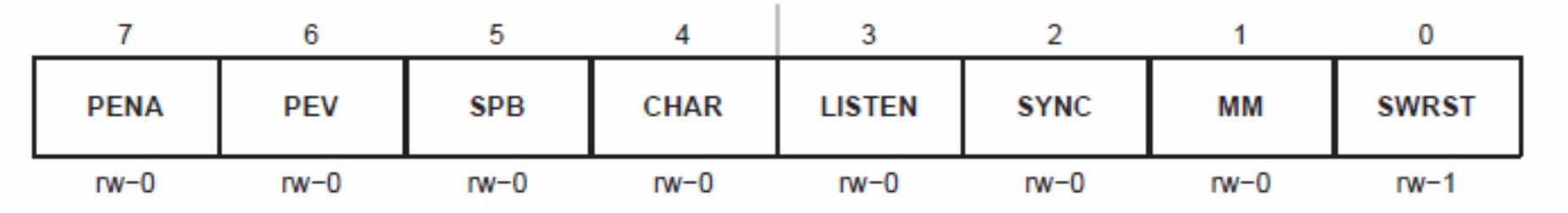
- Por ejemplo: el registro U0CTL (Control de USART0) está ubicado en la dirección **0x070**, es de 8 bits y tiene la siguiente estructura:

7	6	5	4	3	2	1	0
PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–1



7	6	5	4	3	2	1	0
PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

```
__no_init volatile union
{     unsigned char U0CTL;
      struct
      {     unsigned char SWRST : 1;
            unsigned char MM    : 1;
            unsigned char SYNC  : 1;
            unsigned char LISTEN: 1;
            unsigned char CHAR  : 1;
            unsigned char SPB   : 1;
            unsigned char PEV   : 1;
            unsigned char PENA  : 1;
      };
} @ 0x0070;  // mapea la variable a la dir 70h
```



7	6	5	4	3	2	1	0
PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

```
// Configuración de USART0 como UART,
// 8 bits, paridad par, un bit de stop

U0CTL = 0; // pone todo a cero

                // setea los bits necesarios
CHAR = 1;       // 8 bits
PENA = 1;       // paridad habilitada
PEV  = 1;       // paridad par
```

Estructuras y Uniones anónimas

- También son de utilidad para definir variables booleanas, ahorrando memoria.
  - Para ello se usan estructuras de campos de bits. Por ejemplo:

```
union {
  struct {
    unsigned int waitingSomeEvent: 1;
    unsigned int ready4something : 1;
    ......
    unsigned int someEvent       : 1;
    unsigned int anotherFlag     : 1;
  };
  unsigned int flags;
};
```

Estructuras y Uniones anónimas

- Poner a cero todos los flags:

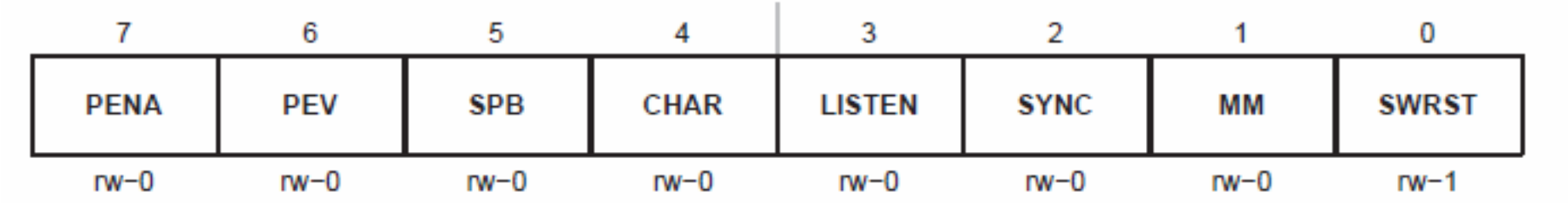
```
flags = 0;
```

- Modificar o testear algún flag:

```
if( waitingSomeEvent && someEvent )
{
    someEvent = 0;
}
```

Manejo de registros mapeados en memoria usando máscaras

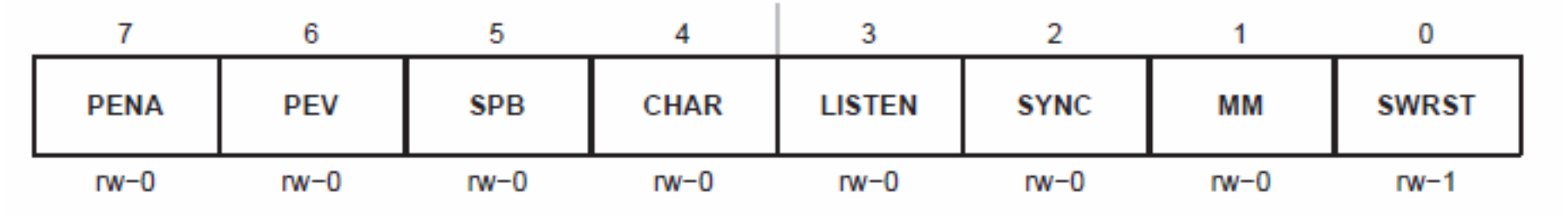
- Para el mismo ejemplo del registro U0CTL:



7	6	5	4	3	2	1	0
PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

```
#define U0CTL (*(unsigned char *)0x0070)
#define SWRST 0x01
#define MM    0x02
......
#define CHAR  0x10
......
#define PENA  0x80
```

Manejo de registros mapeados en memoria usando máscaras



7	6	5	4	3	2	1	0
PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

- Ej. poner a 1 CHAR y MM, el resto en cero:

```
U0CTL = CHAR | MM;
```

- Ej. poner a cero MM sin modificar el resto:

```
U0CTL &= ~MM;
```

Alineamiento de estructuras

- En el MSP430 (y otros micros de 16 bits), los datos de tamaño mayor que un byte deben estar alineados en direcciones pares.
    - En los de 32 bits, se alínean en direcciones múltiplos de 4.

- El compilador agrega bytes de relleno para alinear variables

- También se hace alineamiento dentro de las estructuras.

- El acceso a variables desalineadas requiere de más instrucciones de máquina.

Alineamiento de estructuras

- Cuando no es admisible el relleno dentro de una estructura (por ej. para respetar el formato de un mensaje), se usa la directiva ***#pragma pack***

- El acceso a una variable desalineada genera más código de máquina y es más lento.

Alineamiento de estructuras

- Por ejemplo, dada la siguiente estructura:

```
struct var{
   char c;    // entre c y n1
   int n1;    // un byte de relleno
   int n2;
}v;
```

Veamos la operación `v.n1 = w;` con y sin alineamiento

Alineamiento de estructuras

- Con alineamiento de a 16 bits (por default):

```
mov.w     R10, &0x228
```

- Con alineamiento por byte (*#pragma pack(1)*):

```
mov.b    R10,R14
swpb     R10
and.w    #0xFF,R10
mov.b    R14,&0x24D   ; LSB a dir impar
mov.b    R10,&0x24E   ; MSB a dir par
```

Paso de parámetros a funciones

El paso de parámetros a funciones consume tiempo y puede consumir memoria de la pila.

Dependiendo de la arquitectura del procesador y del compilador, los parámetros se pueden pasar en registros y/o a través de la pila.

En los casos siguientes –sobre MSP430– se ven ambas situaciones.

Paso de parámetros a funciones

Una función con hasta 4 parámetros los recibe en registros (R12, R13, R14 y R15).

Ejemplo:

```
void f2(unsigned char b, int n)
{
  int a, i;
  for(i=0, a=0; i<n; ++i)
    a += b+i;
}
```

Paso de parámetros a funciones

La llamada a ésta función queda así:

En C:

```
f2('a',32);
```

En lenguaje ensamblador:

```
mov.w    #0x20,R13   ; 32
mov.b    #0x61,R12   ; 'a'
call     #f2
```

Paso de parámetros a funciones

Una función con más de 4 parámetros recibe 4 en registros (R12, R13, R14 y R15) y el resto en la pila.

Ejemplo:

```
int f5(int a,int b,int c,int d,int e)
{
  int suma = a + b + c + d + e;

  return suma;
}
```

Paso de parámetros a funciones

El llamado a ésta función se realiza así:

En C:

```
n = f5(10,20,30,40,50);
```

En lenguaje ensamblador:

```
push.w   #0x32          ;el 5° a la pila
mov.w    #0x28, R15 ;4° al 1° en reg.
mov.w    #0x1E, R14
mov.w    #0x14, R13
mov.w    #0x0A, R12
call     #f5
mov.w    R12, R15   ; retorna en R12
```

Paso de parámetros a funciones

Un par de consejos prácticos serían:

- Tratar de usar la mínima cantidad de parámetros que sea posible.

- Analizar si conviene usar variables globales:

  Si bien no es una práctica recomendada en general, en arquitecturas con poca cantidad de RAM puede ayudar a optimizar el uso de memoria y la performance.

- Agrupar variables en estructuras y pasar como parámetro el puntero a la estructura.

Paso de estructuras a funciones

El paso de variables struct, ya sea como parámetro o como valor retornado, conviene hacerlo mediante punteros.

Por ejemplo, dada la siguiente estructura y variables:

```
struct int5
{
     int a, b, d, c, e;
};

struct int5 A, B;
```

Paso de estructuras a funciones

Si en lugar de la función *f5* del ejemplo anterior se implementara mediante otra función *f5s*, que espera un puntero a *struct int5*, en C quedaría así:

```
n = f5s(&A);
```

Y en lenguaje ensamblador:

```
mov.w    #0x220,R12 ; R12: puntero a A
call     #f5s
mov.w    R12,R10    ; R12: valor retorno
```

Paso de estructuras a funciones

Las estructuras son muy útiles porque permiten agrupar variables que tienen relación entre si.

También se las puede aprovechar para optimizar el uso de memoria y la performance.

Requiere un trabajo previo de análisis y diseño, para determinar cuando y donde usarlas.

Paso de estructuras a funciones (por valor o referencia)

Las variables de estructura se pueden pasar como parámetro y ser retornadas por valor y por referencia (puntero).

Sin embargo, el paso por valor implica efectuar una copia, que dependiendo del tamaño de la estructura puede consumir mucho tiempo y memoria.

Veamos dos ejemplos (A y B son de tipo ***struct int5***):

```
B = proc_int5(A);
proc_int5p(&A, &B);   // A: in, B:out
```

Paso de estructuras a funciones (por valor o referencia)

```
B = proc_int5(A);
                mov.w   #0xA,R15
     0x123A:    push.w  0x228(R15)  ; loop
                decd.w  R15         ; copia
                jne     0x132A      ; a pila
                mov.w   #0x234,R12  ; dir B
                call    #proc_int5
proc_int5p( &A, &B );
                mov.w   #0x234,R13  ; &B
                mov.w   #0x22A,R12  ; &A
                call    #proc_int5p
```

Paso de estructuras a funciones (por valor o referencia)

Suponiendo –por ejemplo- que ambas funciones hacen el mismo procesamiento sobre la variable recibida, en el caso:

```
B = proc_int5(A);
```

Se hace una copia a la pila para que la función reciba el contenido de A y se hace otra copia desde la pila a B.

En el caso:

```
proc_int5p(&A, &B);   // A: in, B:out
```

Sólo se efectúa una sola copia, de A a B, luego de procesar.

Fijar la dirección ocupada por una variable

A fin de poder manipular SFR mediante nombres de variables en C, se le puede indicar al compilador la dirección de memoria que debe ocupar una variable.

Por ejemplo:

- En 0x0020 está el registro de entrada del puerto P1
- En 0x0021 está el registro de salida del puerto P1

```
__no_init volatile
unsigned char P1IN @ 0x020,
              P1OUT @ 0x021;
```

Fijar la dirección ocupada por una variable

`__no_init` indica que –a diferencia de otras variables globales o static- estas variables no deben ser creadas con un valor inicial.

una variable `volatile` puede cambiar de valor independiente del programa, por lo que no se puede asumir que su valor no ha cambiado desde el último acceso.

Siempre deben ser leídas desde la memoria.

No se pueden implementar sobre registros.

Otras pautas para escribir código eficiente

- Evitar usar largas cadenas de llamados a funciones
- No usar funciones recursivas
    - Para evitar consumir espacio de la pila en exceso

- Evitar tomar la dirección de variables locales (operador &)
    - La variable deberá ubicarse en memoria (pila) y no en un registro.
    - El optimizador no puede asumir que la variable no será afectada cuando se pasa su dirección a otra función.

Otras pautas para escribir código eficiente

- Cuidados con la operación de negación de bits

Por ejemplo:

```
void f1(unsigned char c1)
{
   if (c1 == ~0x80)
         ..... ;          // Hacer algo
}
```

0x80 es entero de 16 bits → ~0x80 == 0xFF7F

c1 es entero de 8 bits sin signo, para comparar es promovido a 16 bits: 0x007F → c1 != 0xFF7F

Otras pautas para escribir código eficiente

- Para evitar el error debido a la promoción a 16 bits, la solución sería efectuar un *cast* a 8 bits:

```
void f1(unsigned char c1)
{
   if (c1 == (unsigned char)~0x80)
         ..... ;          // Hacer algo
}
```

De esta manera, 0xFF7F se trunca a los 8 MSB, quedando 0x7F y se hace la comparación en 8 bits

Protegiendo el acceso a variables compartidas

- Cuando hay variables compartidas entre procesos asincrónicos (por ejemplo: programa principal y rutina de interrupción), además de ser declaradas `volatile`, puede ser necesario que su actualización se realice en forma atómica (indivisible).

- La directiva `__monitor` precediendo a una función, instruye al compilador para que la misma no pueda ser interrumpida durante su ejecución.

Funciones para atención de interrupciones

- Es muy habitual en la programación de sistemas embebidos tener que atender distintos tipos de interrupciones.

- Cada arquitectura tiene sus propios mecanismos para el manejo y atención de interrupciones:
  - Determinadas direcciones de memoria donde ubicar las primeras instrucciones de cada rutina de interrupción (por ej.: 8051 y derivados, ADSP SHARC, entre otros).
  - Una tabla de vectores de interrupción. Cada vector es un puntero a una rutina de interrupción (MSP430 entre otros)

Funciones para atención de interrupciones

- Las funciones para atención de interrupciones se declaran con la palabra clave `__interrupt`

- Se diferencian del resto de las funciones en que, como se pueden ejecutar en cualquier momento, al entrar deben guardar en la pila no sólo la dirección de retorno, sino también el estado del procesador, a fin de restaurar el estado original al salir.

  - Las demás funciones sólo necesitan guardar en la pila la dirección de retorno.

Funciones para atención de interrupciones

- Ejemplo (para MSP430):

```
#pragma vector=0x0A // Asocia la función
                    // a una interrupc.
                    // determinada

__interrupt void rutina_interrup(void)
{
    /* Hace algo */
}
```

Funciones para atención de interrupciones

En el MSP430 los vectores de interrupción están mapeados a partir de la dirección 0xFFE0 y cada uno ocupa 2 bytes.

El número de vector en la directiva #pragma indica el offset a partir de 0xFFE0 en que está el vector.

```
#pragma vector=0x0A   // 0xFFEA
                      // Interrup.
                      // TimerA1
```

Fijar la dirección inicial de una función

Otra posibilidad a tener en cuenta es la de fijar en que segmento de memoria debe ser ubicada una función.

Dependiendo del compilador será la directiva a usar.

Por ejemplo:

IAR EW430 : `#pragma location="SEGCOD2"`

TI cl430 :

```
#pragma CODE_SECTION(funcA, "SEGCOD2")
```

Fijar la dirección inicial de una función

En ciertos casos puede ser de utilidad que la dirección de determinadas funciones sea conocida y siempre la misma.

Un caso, a modo de ejemplo, es cuando está previsto que el mismo micro pueda actualizar sobre la marcha el programa almacenado en flash.

Fijar la dirección inicial de una función

El procedimiento podría ser:

a) Descargar la nueva versión (y almacenarla en alguna porción libre de la misma flash o en otra memoria.

b) Reemplazar el contenido de la flash

c) Reiniciar el micro con el nuevo programa.

Fijar la dirección inicial de una función

- La función encargada de reemplazar el contenido de la flash debería estar siempre en la misma dirección, para que todas las versiones puedan llamarla.
- El siguiente es un ejemplo (para compilador IAR):

```
#pragma location="CODE3"

void flashPgm()     // Regraba la flash
{                   // y reinicia el
 ......             // micro
```

Fijar la dirección inicial de una función

- En el ejemplo anterior, la memoria flash del MSP430F149 se había estructurado de la siguiente manera:

```
CODE  : 0x1100 a 0xF9FF // programa de
                        // aplicación

CODE2: 0xFA00 a 0xFBFF // framRead()

CODE3: 0xFC00 a 0xFDFF // flashPgm()
```

Fijar la dirección inicial de una función

- Las dos funciones que mantienen inalterable su dirección inicial son:

  – `framRead()` → para leer el nuevo programa almacenado temporalmente en una FRAM

  – `flashPgm()` → lee el nuevo programa de la FRAM, reprograma el segmento CODE y reinicia el el micro.

    Se reprograma toda la flash (incluido los vectores de interrupción), excepto CODE2 y CODE3.

¡ Muchas gracias !

Espero que haya sido de interés y utilidad

Mg. Guillermo Friedrich
UTN-FRBB
gfried@frbb.utn.edu.ar