Programación en C de la tarjeta ADAPT912DT60

La tarjeta ADAPT912DT60 contiene un MC68HC912DT60 (Motorola 68HC12).

La *GNU Development Chain for 68HC11&68HC12* es una iniciativa de Software Libre (SL) que ha añadido la compilación cruzada para los procesadores de la familia 68HC11 y 68HC12 al compilador de C de GNU (GNU-cc). Con esta herramienta es posible, dado un código fuente escrito en C, generar código máquina de microprocesador correspondiente.

El compilador se encarga de:

- Generar código: dada una instrucción en C, generar las instrucciones de código máquina necesarias para implementar la operación.
- Gestión de variables: reservar el espacio y generar las instrucciones necesarias para la gestión de las variables, tanto las globales al programa como las locales a cada una de las funciones.
- Llamadas a funciones: organiza las estructuras en pila para la llamada y regreso de funciones, el paso de parámetros y el retorno de valores
- Definir los tamaños de los distintos tipos. En este caso los tipos tamaños son

Tipo	Número de bytes
`char`	1
`short`	2
`int`	2
`long`	4

1.1 Instalación

Éste compilador está disponible de manera nativa en los sistemas GNU/Linux (y a través del entorno Cygwin en los sistemas Windows).

Los paquetes que es necesario instalar son:

- `gcc-m68hc1x`
- `binutils-m68hc1x`

El primero contiene el compilador y el segundo una serie de utilidades para manejar los ficheros objeto y ejecutable. Estos paquetes se pueden descargar del aula virtual de la asignatura ya que no suelen estar disponibles en los repositorios de las últimas versiones de las distintas distribuciones, pero (hasta el momento) se pueden instalar sin incompatibilidades.

Una vez escrito el código fuente (fichero `.c`) el procedimiento para poder ejecutar el programa se describe en los siguientes apartados. La forma más adecuada de realizar esta secuencia es escribir un fichero `Makefile` y hacer uso de la herramienta `make`.

1.1.1 Compilación

Proceso para obtener el fichero objeto (`.o`) a partir del fichero fuente. Esto se hace con el siguiente comando:

```
m68hc11-gcc -m68hc12 -mshort  -DADAPT_RAM28  -c -o fichero.o fichero.c
```

Se invoca al compilador y se le indica que la CPU destino es la de un 68HC12 (`-m68hc12`) y que el programa se va a ejecutar es la tarjeta Adapt912dt60 en el bloque RAM de 28Kb (`-DADAPT_RAM28`).

1.1.2 Enlazado

Con este proceso se combinan uno o varios ficheros objetos para dar lugar a un fichero en formato `elf`. El comando tiene la siguiente forma

```
m68hc11-gcc -m68hc12 -mshort -Wl,-m,m68hc12elfb -o fichero.elf fichero.o
```

donde es necesario indicar, al igual que antes, que se enlaza para un 68HC12 (`-m68hc12`) y se indica un fichero de configuración con la disposición que tendrán los distintos bloques de memoria (`-Wl,-m,m68hc12elfb`) en el momento de la ejecución.

El contenido del fichero `elf` se puede ver con el comando `m68hc12-objdump` con el que se puede obtener la dirección y el contenido de las distintas secciones, el valor definitivo de todas las etiquetas y el desensamblado de nuestro código.

1.1.3 Obtención del fichero s19

En un sistema GNU/Linux en PC, es directamente el fichero en formato `elf` el que se carga en memoria y se ejecuta. Pero en el caso de los microcontroladores de la familia 68HC1X se utiliza un fichero en formato `S19`, que fue definido por el fabricante, para el envío del ejecutable al microcontrolador. Éste recibirá el fichero por la línea serial y lo cargará en memoria para su posterior ejecución.

Por ello es necesario convertir el fichero `.elf` en fichero `.s19`, para lo que se utiliza el comando:

```
m68hc11-objcopy --output-target=srec  --srec-len 32  \
  --only-section=.text --only-section=.rodata --only-section=.vectors \
  --only-section=.data \
  fichero.elf fichero.s19
```

El formato de los ficheros `S19` también puede inspeccionarse. Estos son ficheros de texto en los que cada línea comienza por los caracteres `S0`, `S1` o `S9`. La línea `S0` es la primera del fichero, y la `S9` la última. Las líneas `S1` son las que contienen los bytes a depositar en memoria y tiene la siguiente estructura:

- `S1` primeros 2 caracteres
- 2 caracteres hexadecimales que indican el número total de bytes hasta final de línea (fin del registro)
- 4 caracteres hexadecimales que contienen la dirección de memoria absoluta (2 bytes) a partir de la cual se deben depositar los siguientes bytes del registro
- grupos de 2 caracteres que codifican en hexadecimal los bytes que hay que depositar en memoria.
- Un último byte (2 caracteres hexadecimales) que corresponde al *checksum* de comprobación del registro (línea).

1.2 Librerías

La mayoría de las funcionalidades que asociamos al lenguaje C realmente no dependen del compilador sino de las librerías asociadas. Por ejemplo las funciones de entrada salida (`open`, `write`, `printf`, `scanf`, ...), dependen de librerías (`stdio`, `unistd`, ...) y no del compilador.

En sistemas de propósito general se han estandarizado como parte del ANSI-C ciertas librerías y son distribuidas junto con el compilador, por lo que esas funciones básicas estarán siempre

disponibles.

En cambio, en los microcontroladores no suele existir esa disponibilidad, principalmente porque en muchos de ellos no tienen sentido muchas de esas funciones. Por ejemplo:

- hacer `printf` para sacar por pantalla un mensaje en un sistema empotrado que sólo tiene leds.
- hacer `write` sobre un fichero en un sistema empotrado que no tiene sistema de archivos ya que no tiene otro almacenamiento que no sea la memoria principal.

Para ciertos microcontroladores con ciertas capacidades hay proyectos que tratan de crear librerías que tengan el mayor número posible de las funciones ANSI-C. Un ejemplo es el proyecto Newlib, el cual también está portado para el 68HC11 y 68HC12.

La ventaja de estas librerías es que se tienen disponible, en el sistema empotrado, funciones de C con las que estamos familiarizados. Pero hay, al menos, dos inconvenientes:

- Nunca se consigue la implementación completa de las funciones, por las carencias en el hardware que mencionamos antes. Por ello es necesario consultar en cada caso que opciones están disponible y con que limitaciones.
- La implementación de ciertas funcionalidades en el microcontrolador conlleva un gran tamaño en el código generado y un mayor tiempo de ejecución. Por ejemplo las funciones `printf` y `scanf` se pueden implementar, utilizando como pantalla la línea serial de microcontrolador, pero la gestión de parámetros variables y formatos hace que el código de un programa pase de 4Kb a cerca de los 16Kb por utilizar una de esas funciones en un único punto de nuestro código.

En cambio hay otras iniciativas que tratan de hacer librerías específicamente desarrolladas para ciertas familias de microcontroladores. La librería *GNU Embedded Libraries for 68HC11 and 68HC12* (GEL) está específicamente desarrollada para una serie de placas de desarrollo comerciales que utilizan microcontroladores de la familia 68HC11 y 68HC12.

En esta asignatura se ha optado por utilizar la librería `SimpleGEL`, adaptada de una de las versiones iniciales de la GEL realizando ajustes para tener en cuenta las especificidades de la placa ADAPT912DT60, la cual no venía contemplada inicialmente.

2 La librería SimpleGEL

Las principales funcionalidades de esta librería son las siguientes.

2.1 Tamaños de tipos de datos

Incluyendo el fichero `types.h` quedan definidos (a partir de los tipos disponibles) los tipos estándar `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t` y `uint32_t` que son enteros con y sin signo de un tamaño conocido.

2.2 Acceso a los periféricos

El acceso a los registros de control de los distintos periféricos incluidos en el microcontrolador se realiza a través del *array* de bytes `_io_ports[]` (para lo cual hay que incluir el fichero `sys/ports.h`). Este *array* da acceso a la zona de memoria donde se encuentre el bloque de registros y sigue la distribución de los mismos según figura en la página 61 del *Manual Referencia MC68HC912D60A*.

Para facilitar el acceso a estos registro, están definidas constantes que almacenan la posición dentro de dicho array donde se encuentran los registros según el nombre que el fabricante les da a los mismo. Estas etiquetas comienzan por `M6812_` seguidas del nombre del registro en mayúsculas.

Por ejemplo para acceder al registro PUCR que se encuentra en la posición 12 ($000C) de la tabla de registros podemos utilizar la expresión _io_ports[12] ó _io_ports[0x0C], pero queda mucho más legible si se utiliza _io_ports[M6812_PUCR].

De la misma manera, están definidas etiquetas para el nombre de los bits dentro de los registros de control. En este caso las etiquetas comienzan por M6812B_ seguidas por el nombre del bit. Estas constantes lo que definen es un byte que tiene un 1 en la posición del bit correspondiente y un 0 en el resto de los bits (máscara de bit).

Para poner a 1 el bit PUPG (bit 6) del registro PUCR, y dejar el resto a 0, se puede utilizar la expresión _io_ports[M6812_PUCR]=64 ó _io_ports[M6812_PUCR]=0x40, pero queda más claro poniendo _io_ports[M6812_PUCR]=M6812B_PUPG.

2.2.1 Acceso a registros de doble byte

El *array* _io_ports[] es de tipo uint8_t, es decir 1 byte. Pero ciertos registros, especialmente los del subsistema temporizador, deben ser accedidos como word (doble byte). Para conseguir esto hay que indicarle al compilador que en ese momento el *array* _io_ports[] es de tipo uint16_t.

Para facilitar estos accesos está definida, en el fichero sys/ports.h, la macro _IO_PORTS_W(p) que define un *array* de doble bytes (uint16_t). De esta manera, para copiar el contenido del registro TCNT se puede hacer con la siguiente instrucción:

```
uint16_t tcnt = _IO_PORTS_W(M6812_TCNT);
```

Para cargar un valor en el registro TC2 se puede hacer con la instrucción

```
_IO_PORTS_W(M6812_TC2) = valor;
```

2.3 Operaciones sobre bits de los registros de control

Sobre los bits del los registros de control hay una serie de operaciones que necesitaremos realizar, como son fijar y conocer su valor. Usando adecuadamente los operadores bit a bit del lenguaje C dichas expresiones pueden quedar de una forma breve y fácilmente legible.

2.3.1 Operadores a nivel de bit

Los operadores a nivel de bits del lenguaje C operan con todos los bits de los operandos, pero realiza la operación entre los bits correspondientes de los dos operandos. Es decir, el bit 0 del primer operando se opera con el bit 0 del segundo operando y da lugar al bit 0 del resultado; el bit 1 del primer operando se opera con el bit 1 del segundo operando y da lugar al bit 1 del resultado; y así sucesivamente.

Los operadores de bits son los siguientes:

- o-bit a bit: operador binario que da 0 sólo cuando ambos bits son 0. El operador es la barra vertical | (que se obtiene en el teclado español con <AltGr>+<1>):

```
  0101 0101
| 0011 0011
-------------
  0111 0111
```

  Visto de otra manera, cuando se hace la O de un bit con un 0, el bit queda como está (X | 0= X), mientras que si se hace con un 1 el resultado será siempre 1 (X | 1= 1).

- y-bit a bit: operador binario que da 1 sólo cuando ambos bits son 1. El operador es el *ampersand* &

```
  0101 0101
```

```
& 0011 0011
-------------
  0001 0001
```

Visto de otra manera, cuando se hace la y-lógica de un bit con un 1, el bit queda como está (`X & 1= X`), mientras que si se hace con un 0 el resultado será siempre 0 (`X & 0= 0`).

- o exclusiva-bit a bit: operador binario que da 1 sólo cuando ambos bits son distintos. El operador es el acento circunflejo ^ (en el teclado español se obtiene pulsando la tecla donde aparece el símbolo, junto a la P, y luego pulsando la barra espaciadora)

```
  0101 0101
^ 0011 0011
-------------
  0110 0110
```

Visto de otra manera, cuando se hace la o exclusiva-bit a bit de un bit con un 0, el bit queda como está (`X ^ 0= X`), mientras que si se hace con un 1 el resultado será en bit invertido (`X ^ 0= ~X`).

- no-bit a bit: operador monario (un solo operando) que devuelve el bit invertido. El operador es la tilde ~ (en el teclado español se obtiene con <AltGr>+<ñ>

```
~ 0011 0011
-------------
  1100 1100
```

2.3.2 Operadores de desplazamiento

Los operadores de desplazamiento de bits devuelve como resultado el valor del operando de su izquierda, pero con sus bits desplazados a la derecha/izquierda el número de posiciones (n) indicadas por su operando de la derecha. El número de desplazamientos debe ser (se toma) como entero sin signo.

Los operadores de desplazamiento son:

- desplazamiento a la izquierda: desplaza los bits a la izquierda el número (`n`) de posiciones indicadas. Los `n` bits superiores se pierden y por la derecha *entran* `n` ceros. El operador son dos *mayor-que* seguidos **<<**:

```
0101 0101 << 1 =  1010 1010
0101 0101 << 2 =  0101 0100
0101 0101 << 3 =  1010 1000
0101 0101 << 4 =  0101 0000
```

- desplazamiento a la derecha: desplaza los bits a la derecha el número (`n`) de posiciones indicadas. Los `n` bits inferiores se pierden. Por la izquierda *entran* `n` ceros, siempre y cuando operando de la izquierda sea sin signo (que en general es lo que querremos). El operador son dos *menor-que* seguidos **>>**:

```
0101 0101 >> 1 =  0010 1010
0101 0101 >> 2 =  0001 0101
0101 0101 >> 3 =  0000 1010
0101 0101 >> 4 =  0000 0101
```

2.3.3 Máscaras

Para referenciar a los bits que nos interesan se podrían definir constantes con el número de bit. Por ejemplo:

```
#define POS_M6812B_PUPG  7
#define POS_M6812B_PUPH  8
#define POS_M6812B_PUPA  0
```

Pero la representación más útil es una en la que se defina un entera que tenga un 1 en la posición del bit correspondiente y 0 en el resto de los bits. A esto se le llama una **máscara de bit**.

Si tenemos lo anterior, para obtener la máscara bastará con definir:

```
#define M6812B_PUPG  (1 << POS_M6812B_PUPG)
  equivale a (1 << 6) que es 0100 0000
#define M6812B_PUPH  (1 << POS_M6812B_PUPH)
  equivale a (1 << 7) que es 1000 0000
#define M6812B_PUPA  (1 << POS_M6812B_PUPA)
 equivale a (1 << 0) que es 0000 0001
```

2.3.4 Operaciones Habituales

Vamos a ver ahora una serie de operaciones que querremos hacer, dado un registro y unas máscaras de bits.

A) Puesta a 1 de un bit y a 0 el resto

Como la máscara definida para el nombre de un bit tiene, precisamente, esa configuración, basta con asignar la etiqueta correspondiente al registro.

Ejemplo: Poner a 1 el bit PUPG del registro PUCR, y el resto a 0

```
_io_ports[M6812_PUCR] = M6812B_PUPG;
```

B) Puesta a 1 de varios bits y a 0 el resto

Si son varios los bits a poner a 1, hay que hacer la o-bit a bit de las máscaras que definen los bits que se quieren poner a 1 utilizando la barra vertical (|).

Ejemplo: Poner a 1 los bit PUPG y PUPB del registro PUCR, y el resto a 0

```
_io_ports[M6812_PUCR] = M6812B_PUPG | M6812B_PUPB;
```

C) Puesta a 0 de un bit y el resto a 1

Es asignar el valor invertido (no-lógico) del caso A anterior. Habrá que aplicarle la tilde (~) a la etiqueta antes de hacer la asignación.

Ejemplo: Poner a 0 el bit PUPG del registro PUCR, y el resto a 1

```
_io_ports[M6812_PUCR] = ~ M6812B_PUPG;
```

D) Puesta a 0 de varios bits y a 1 el resto

Si son varios los bits a poner a 0, hay dos formas de hacerlo: hacer la o-bit a bit de las etiquetas y luego invertir; o invertir las etiquetas y luego hacer la y-bit a bit.

Ejemplo: Poner a 0 los bit PUPG y PUPB del registro PUCR, y el resto a 1

```
_io_ports[M6812_PUCR] = ~(M6812B_PUPG | M6812B_PUPB);
```

o de la segunda forma (ley de Morgan):

```
_io_ports[M6812_PUCR] = ~M6812B_PUPG & ~M6812B_PUPB;
```

E) Puesta a 1 de un bit dejando los demás como están

Para poner en un registro a 1 un bit y dejar los demás como están se debe utilizar la operación o-bit a bit entre el valor actual y una máscara que tenga un 1 en el bit que se desea que pase a 1, y 0 en el resto. De esta manera en el resultado el bit deseado pasará a 1 y el resto quedarán como estaba en el registro. Ese resultado debe asignarse al registro.

Ejemplo: Poner a 1 el bit PUPG del registro PUCR, y dejar el resto como están

```
_io_ports[M6812_PUCR] = _io_ports[M6812_PUCR] | M6812B_PUPG;
```

Podemos hacer uso del operador o-bit a bit y asignación (|=) para tener una expresión más compacta:

```
_io_ports[M6812_PUCR] |= M6812B_PUPG;
```

F) Puesta a 1 de varios bits dejando los demás como están

Siguiendo la lógica del apartado anterior, la máscara en esta ocasión tendrá un 1 por cara bit que se deba convertir a 1. Esto se consigue haciendo la o-bit a bit de las máscaras de los bits.

Ejemplo: Poner a 1 el bit PUPG y el PUPB del registro PUCR, y dejar el resto como están

```
_io_ports[M6812_PUCR] = _io_ports[M6812_PUCR] | M6812B_PUPG | M6812B_PUPB;
```

Podemos hacer uso del operador o-bit a bit y asignación (|=) para tener una expresión más compacta:

```
_io_ports[M6812_PUCR] |= (M6812B_PUPG | M6812B_PUPB);
```

G) Puesta a 0 de un bit dejando los demás como están

Para poner en un registro a 0 un bit y dejar los demás como están se debe utilizar la operación y-bit a bit entre el valor del registro y una máscara que tenga un 0 en el bit que se desea que pase a 0, y 1 en el resto. Esa máscara se consigue invirtiendo la máscara del nombre del bit.

Ejemplo: Poner a 0 el bit PUPG del registro PUCR, y dejar el resto como están

```
_io_ports[M6812_PUCR] = _io_ports[M6812_PUCR] & ~M6812B_PUPG;
```

Podemos hacer uso del operador y-bit a bit y asignación (&=) para tener una expresión más compacta:

```
_io_ports[M6812_PUCR] &= ~M6812B_PUPG;
```

H) Puesta a 0 de varios bits dejando los demás como están

Siguiendo la lógica del apartado anterior, la máscara en esta ocasión tendrá un 0 por cara bit que se deba convertir a 0. Esto se consigue haciendo la y-bit a bit de las etiquetas de los bits invertidas.

Ejemplo: Poner a 0 el bit PUPG y el PUPB del registro PUCR, y dejar el resto como están

```
_io_ports[M6812_PUCR] = _io_ports[M6812_PUCR] & ~M6812B_PUPG & ~M6812B_PUPB;
```

Podemos hacer uso del operador y-bit a bit y asignación (&=) para tener una expresión más compacta:

```
_io_ports[M6812_PUCR] &= ~M6812B_PUPG & ~M6812B_PUPB;
```

o alternativamente (ley de Morgan):

```
_io_ports[M6812_PUCR] &= ~ (M6812B_PUPG | M6812B_PUPB);
```

I) Invertir el valor de un 1 bit

Dada las propiedades de la o exclusiva-bit a bit, para invertir un bit basta con aplicar una máscara donde esté a 1 el bit que queramos invertir.

Ejemplo: Poner invertir el bit 7 del puerto G

```
_io_ports[M6812_PORTG] = _io_ports[M6812_PORTG] ^ M6812B_PG7;
```

Podemos hacer uso del operador o exclusiva-bit a bit y asignación (`^=`) para tener una expresión más compacta:

```
_io_ports[M6812_PORTG] ^= M6812B_PG7;
```

J) Actuar si un bit está a 1

Si queremos ejecutar un código (`if` o `while`) si un bit está a 1 tenemos que aplicar y-bit a bit con una máscara para conservar únicamente el bit de interés. Si éste está a 1 la condición será verdad (distinto de 0) y si está a 0 la condición será falsa (igual a 0). La máscara del bit es la adecuada.

Ejemplo: Ejecutar algo si está a 1 el bit 5 del puerto A

```
if(_io_ports[M6812_PORTA] & M6812B_PA5) { ... }
```

K) Actuar si un bit está a 0

Si queremos ejecutar un código (`if` o `while`) si un bit está a 0 tenemos que utilizar el método anterior e invertir el resultado lógico con el operador no-lógico (`!`) .

Ejemplo: Ejecutar algo si está a 0 el bit 5 del puerto A

```
if(!(_io_ports[M6812_PORTA] & M6812B_PA5)) { ... }
```

L) Actuar si varios bits están en ciertos valores

Si queremos ejecutar un código (`if` o `while`) si ciertos bit están a 1 o 0, lo más sencillo es concatenar las condiciones anteriores, según corresponda, utilizando la y-lógica de circuito corto (&&). De esta manera se pueden comprobar bits de distintos registros.

Ejemplo: Ejecutar algo si está a 1 el bit 5 del puerto A y a 0 el bit 3 del puerto B

```
if((_io_ports[M6812_PORTA] & M6812B_PA5)
      && !(_io_ports[M6812_PORTB] & M6812B_PB3)) { ... }
```

Si son varios bits del mismo puerto es más eficiente aplicar una máscara para quedarse con los bits de interés y ver si el resultado tiene el valor deseado.

Ejemplo: Ejecutar algo si está a 1 el bit 5 del puerto A y a 0 el bit 3 del puerto A

```
if( ( _io_ports[M6812_PORTA] & (M6812B_PA5 | M6812B_PB3) )
      == M6812B_PB5 ) { ... }
```

M) Recorrer los bits de un registro

Si queremos recorrer los bits de un registro para hacer alguna operación sobre ellos (por ejemplo saber cuantos están a 1) habría dos opciones.

Una primera es ir creando máscara para cada bit e irla aplicando sucesivamente.

Ejemplo: Saber cuantos bits están a 1 en `registro`

```
int bitsAUno = 0;
for (int bitActual = 0; bitActual < totalBits; bitActual++)
  if (registro & (1 << bitAcatual))
```

```
    bitsAUno++;
```

Una segunda opción es considerar solo en bit 0 e ir desplazando los bits hacia la derecha. Este método tiene la ventaja de que, como van entrando ceros por la izquierda, terminaríamos en cuanto lo que quede en registro sea 0.

Ejemplo: Saber cuantos bits están a 1 en `registro`

```
int bitsAUno = 0;
while (registro) {
  if (registro & 1)
    bitsAUno++;
  registro >>= 1;
}
```

N) Construir el valor de un registro

Si queremos generar una determina configuración de bits en un registro según una condición general (por ejemplo poner a 1 los bits en posiciones múltiplo de 3), tendremos que realizar el procedimiento inverso al anterior.

Una primera opción es ir creando la máscara para cada bit e irla aplicando sucesivamente.

Ejemplo: Poner 1 los bits en posiciones múltiplo de 3 y el resto a 0 en `registro`

```
registro = 0;  // inicialmente todos los bits a 0
for (int bitActual = 0; bitActual < totalBits; bitActual += 3)
  registro |= (1 << bitAcatual);
```

Una segunda opción es considerar solo en bit 0 e ir desplazando los bits hacia la izquierda. En este caso, debemos empezar por los bits más significativos (los que quedarán más a la izquierda).

Ejemplo: Poner 1 los bits en posiciones múltiplo de 3 y el resto a 0 en `registro`

```
registro = 0;  // inicialmente todos los bits a 0
for (int bitActual = totalBits - 1; bitActual >= 0; bitActual--) {
  if (!(bitActual % 3))
    registro |= 1;
  registro <<= 1;
}
```

2.4 Entrada y salida serial

Un periférico muy útil, y por lo tanto muy utilizado en cualquier desarrollo, es la comunicación serial SCI. Esta permite enviar mensajes de depuración y recibir datos desde un PC con una simple conexión serial.

Se puede manejar, como el resto de los periféricos, a través de los registros de control, pero para facilitar su uso están disponibles una serie de funciones sencillas para su manejo. Es necesario incluir el fichero `sys/sio.h` para tener acceso a las mismas.

2.4.1 Funciones

La lista de funciones disponibles son:

- `void serial_init(void)` inicializa la serial y la configura a la velocidad indicada por la constante `M6812_DEF_BAUD`, definida en el fichero `sys/params.h`

- `int serial_receive_pending(void)` devuelve != 0 si hay caracteres pendientes de leer.
- `void serial_flush (void)` espera hasta se terminen de enviar los caracteres.
- `void serial_send(char)` envía el carácter por la línea serial.
- `char serial_recv(void)` espera recepción de un carácter y lo devuelve.
- `void serial_print(const char*)` envía cadena de caracteres por la serial.
- `void serial_getline(char*)` espera la recepción de una línea por la serial. `buf` debe estar definida y ser de tamaño necesario.
- `uint8_t serial_getbinbyte()` espera 8 bits (0s o 1s) por la serial para formar un byte. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `void serial_printbinbyte(uint8_t)` Saca por la serial la configuración de 8 bits (0s y 1s) del byte pasado
- `void serial_printbinword(uint16_t)` Saca por la serial la configuración de 16 bits (0s y 1s) del doble byte pasado .
- `uint8_t serial_gethexbyte()` Espera 2 dígitos hexadecimales para formar byte. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `utin16_t serial_gethexword()` Espera 4 dígitos hexadecimales para formar doble byte. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `void serial_printhexbyte(uint8_t)` Saca por la serial, en hexadecimal, el byte pasado.
- `void serial_printhexword(uint16_t)` Saca por la serial, en hexadecimal, el doble byte pasado.
- `uint8_t serial_getdecbyte()` Espera por la serial dígitos decimales para formar byte sin signo. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `uint16_t serial_getdecword()` Espera por la serial dígitos decimales para formar doble byte sin signo. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `uint32_t serial_getdeclong()` Espera por la serial dígitos decimales para formar cuádruple byte sin signo. Permite borrar y termina cuando el usuario pulsa el salto de línea.
- `void serial_printdecbyte(uint8_t)` Saca por la serial, en decimal, el byte pasado.
- `void serial_printdecword(uint16_t)` Saca por la serial, en decimal, el doble byte pasado.
- `void serial_printdeclong(uint32_t)` Saca por la serial, en decimal, el cuádruple byte pasado.

2.5 Gestión de interrupciones

La mayoría de los periféricos de este microcontrolador pueden generar interrupciones. La activación y enmascaramiento de las distintas interrupciones se debe hacer a través de los bits de los correspondientes registros de control.

Si se va a hacer uso de interrupciones se debe incluir el fichero `sys/interrupts.h`

2.5.1 Habilitación

La habilitación y deshabilitación general de las interrupciones enmascarables se hace con las funciones `unlock()` y `lock()` respectivamente. Éstas se utilizan, normalmente, durante la inicialización del sistema (primera parte de la función `main`).

Recordar que no es necesario utilizarlas en las rutinas de tratamiento de interrupciones, ya que estas se deshabilitan automáticamente al llamar la función y se habilitan al salir. Es decir, las funciones se ejecutan con las interrupciones deshabilitadas, por ello la necesidad de que duren lo mínimo posible para evitar el retraso en atender interrupciones de otros periféricos.

2.5.2 Rutinas de tratamiento

La rutina de tratamiento de cada una de las interrupciones se define declarando en nuestro código una función con un nombre determinado. Con esto la librería se encarga de definir el vector necesario en la tabla de vectores de interrupción. El nombre de las funciones, según la interrupción, figura en la siguiente tabla.

Nombre	Interrupción
`vi_osc`	Interrupción del sistema oscilador
`vi_cantx`	Transmisión en el subsistema CAN
`vi_canrx`	Recepción en el subsistema CAN
`vi_canerr`	Errores en el subsistema CAN
`vi_pabov`	Desbordamiento del acumulador de pulsos B
`vi_cmuv`	Cuenta final del contador de módulo
`vi_kwgh`	Activación de los puestos G ó H
`vi_canwu`	Activación del subsistema CAN
`vi_atd`	Conversores analógicos
`vi_sci1`	Interrupción del sistema SCI 1
`vi_sci0`	Interrupción del sistema SCI 0
`vi_spi`	Interrupción del sistema SPI
`vi_pai`	Flanco del acumulador de pulsos
`vi_paov`	Desbordamiento del acumulador de pulsos A
`vi_tov`	Desbordamiento del temporizador
`vi_ioc7`	Canal 7 del temporizador
`vi_ioc6`	Canal 6 del temporizador
`vi_ioc5`	Canal 5 del temporizador
`vi_ioc4`	Canal 4 del temporizador
`vi_ioc3`	Canal 3 del temporizador
`vi_ioc2`	Canal 2 del temporizador
`vi_ioc1`	Canal 1 del temporizador
`vi_ioc0`	Canal 0 del temporizador

Nombre	Interrupción
`vi_rti`	Interrupción de tiempo real
`vi_irq`	Interrupción línea IRQ
`vi_xirq`	Interrupción línea XIRQ
`vi_swi`	Instrucción SWI
`vi_trap`	Instrucción Ilegal
`vi_copreset`	Reset por temporizador COP
`vi_clkreset`	Reset por fallo del reloj
`_start`	Reset externo o reset de encendido. Inicializa el sistema y llama a nuestro `main()`

Además del nombre adecuado, la función debe tener el siguiente prototipo:

```
void __attribute__((interrupt)) nombre(void) { … }
```

Por ejemplo, para definir la función de tratamiento de la interrupción del temporizador 2 el prototipo debe ser:

```
void __attribute__((interrupt)) vi_ioc2(void) { … }
```

Si la función tiene que comunicarse con el programa principal sólo puede hacerlo modificando variables globales, es decir, declaradas al principio del fichero y fuera de cualquier función.

2.5.3 Puertos disponibles

Para no tener que grabar en la *flash* de los 68HC12 en cada prueba de programa (lo cual terminaría con su ciclo de vida) se utiliza una memoria RAM. Los programas en C no caben en los 2Kb de memoria RAM interna disponibles. Por ese motivo se ha añadido a las placas Adapt912dt60, una placa de expansión de memoria de 64Kb.

Durante el arranque, si el pin PS4 está a tierra (existe un *jumper* entre los pines 1 y 50 del conector H1) el firmware de arranque habilita el modo extendido y coloca 28Kb de RAM externa en el mapa de memoria, que será utilizado para cargar los programas C realizados. Los puertos A, B y E son utilizados para el bus externo, por lo que no pueden utilizarse en las aplicaciones. El resto de puertos del conector H2 (puerto P, CAN y analógico 1) sí están disponibles cuando la tarjeta de memoria tiene los correspondientes pines macho del conector H2.