Práctica 1: Introducción al entorno de desarrollo

Prof. Alberto A. Del Barrio

1 Objetivos

El componente principal del equipo que vamos a utilizar en el laboratorio es un microcontrolador ARM. En esta primera práctica trataremos de familiarizarnos con la arquitectura de este procesador y su programación. En concreto, los objetivos que perseguimos en esta práctica son:

- Familiarizarse con el entorno de desarrollo Eclipse y las herramientas GNU para ARM, ensamblador y enlazador esencialmente.
- Comprender la estructura de un programa y el proceso de generación de un código binario a partir de éste.
- Estudiar el sistema de E/S de la placa de desarrollo S3CEV40, así como el manejo de periféricos sencillos por medio de interrupciones.

2 La Arquitectura ARM

Los procesadores de la familia ARM tienen una arquitectura RISC de 32 bits, ideal para realizar sistemas empotrados de elevado rendimiento y reducido consumo, como teléfonos móviles, PDAs, consolas de juegos portátiles, etc. Tienen las características principales de cualquier RISC:

- Un banco de registros.
- Arquitectura load/store, es decir, las instrucciones aritméticas operan sólo sobre registros, no directamente sobre memoria
- Modos de direccionamiento simples. Las direcciones de acceso a memoria (load/store) se determinan sólo en función del contenido de algún registro y el valor de algún campo de la instrucción (valor inmediato).
- Formato de instrucciones uniforme. Todas las instrucciones ocupan 32 bits con campos del mismo tamaño en instrucciones similares.

La arquitectura ARM sigue un modelo Von Neumann con un mismo espacio de direcciones para instrucciones y datos. Esta memoria es direccionable por byte y está organizada en palabras de 4 bytes.

En modo usuario son visibles 15 registros de propósito general (R0-R14), un registro contador de programa (PC), que se denomina también para esta arquitectura como R15, y un registro de estado (CPSR). Todas las instrucciones pueden direccionar y escribir cada uno de los 16 registros en cada momento. El registro de estado, CPSR, debe ser manipulado por

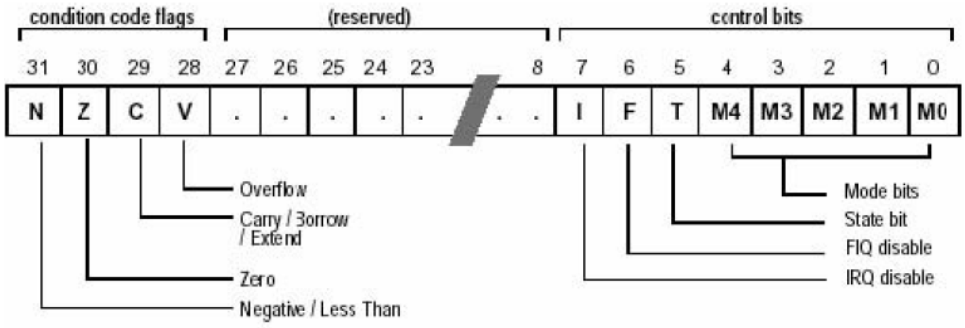


Figura 1: Registro de Estado del ARM (CPSR)

medio de instrucciones especiales.

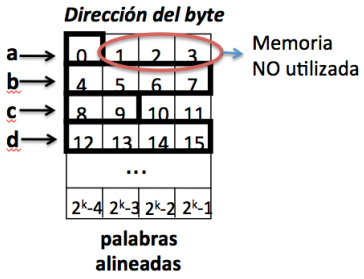
El registro de estado, CPSR, almacena información adicional necesaria para determinar el estado del programa, por ejemplo el signo del resultado de alguna operación anterior o el modo de ejecución del procesador. Es el único registro que tiene restricciones de acceso. Está estructurado en campos con un significado bien definido: flags, extensión (reservados) y control, como ilustra la figura 1. El campo de flags contiene los indicadores de condición y el campo de control contiene distintos bits que sirven para controlar el modo de ejecución.

Algunos bits están reservados para uso futuro, y por tanto, no son modificables (siempre se leen como cero). Los bits N, Z, C y V (indicadores de condición) son modificables en modo usuario, mientras que los otros bits sólo son modificables en los modos privilegiados. Dichos bits tienen el siguiente significado:

- N: Indica si la última operación dio como resultado un valor negativo ($N = 1$) o positivo ($N = 0$).
- Z: Indica si el resultado de la última operación fue cero ($Z = 1$).
- C: Su valor depende del tipo de operacin:
  - Para suma o comparación, $C = 1$ si hubo carry.
  - Para las operaciones de desplazamiento toma el valor del bit saliente.
- V: En el caso de una suma o una resta $V = 1$ indica que hubo overflow.

Además del PC y del CPSR hay una serie de registros con una función especial, por lo que no es recomendable utilizarlos como si fueran de propósito general. Son los siguientes:

- *R9*: puntero base utilizado con bibliotecas dinámicas (SB).
- *R10:* puntero límite de pila (SL).



(a) Memoria alineada

Dirección	Contenido
16	AA
17	BB
18	CC
19	DD

BIG-ENDIAN: byte MÁS significativo primero

(b) Big Endian

Dirección	Contenido
16	DD
17	CC
18	BB
19	AA

LITTLE-ENDIAN: byte MENOS significativo primero

(c) Little Endian

Figura 2: Alineamiento de la memoria y concepto de endianness

- *R11:* puntero marco de pila o *Frame Pointer* (*FP*).
- *R12:* registro auxiliar utilizado en las llamadas a subrutina o *Intra-Procedure scratch* (*IP*).
- *R13:* puntero de pila o *Stack Pointer* (*SP*).
- *R14:* registro de enlace o *Link Register* (*LR*). Se utiliza para almacenar la dirección de retorno en un salto a subrutina.

Especialmente importantes serán los registros R11-R14, como veremos a continuación.

2.1 La memoria

2.1.1 Disposición de los datos

En primer lugar, hemos de recordar que en el ARM, como en la mayorá de las arquitecturas actuales, los datos aparecen alineados en memoria. Dado que las instrucciones son de 32-bits (4 bytes), éstas aparecerán en posiciones múltiplo de 4. Además, hay que tener en cuenta la existencia de datos de tamaño distinto de 4 bytes, en concreto de 1 ó 2 bytes, que aparecerán en posiciones múltiplo de 1 (todas) ó 2, respectivamente. Este hecho se observa en la figura 2a. Suponiendo que declaramos 4 variables *a*, *b*, *c* y *d*, y que *a* es de tipo char (1 byte), *c* de tipo short (2 bytes) y *b* y *d* son de tipo int (4 bytes), puede verse que ciertas posiciones de memoria no son utilizadas a fin de conseguir el alineamiento de las palabras.
Nótese que las estructuras más complejas como *structs* y *arrays* también respetan el alineamiento, al alinear cada uno de los elementos que los componen.

Además del alineamiento, los datos se organizan de acuerdo al formato de *endianness*. La arquitectura del ARM permite los dos tipos de endianness:

- *Big endian.* El byte más significativo de la palabra ocupa la dirección más baja, es decir, se lee primero.

- *Little endian.* El byte menos significativo de la palabra ocupa la dirección más baja, es decir, se lee primero.

Suponiendo que la palabra en hexadecimal 0xAABBCCDD está almacenada en la posición 16 de memoria, ambos endiannes pueden observarse en las figuras 2b y 2c. En nuestro caso, seleccionaremos little endian al configurar los proyectos.

2.1.2 Regiones de memoria

El estándar de memoria está diseñado para programas mono-hilo o mono-proceso (en nuestro caso consideraremos ambos términos intercambiables). Cada proceso tiene un estado definido por el contenido de los registros arquitectónicos y por el contenido de la memoria que puede direccionar. De acuerdo al estándar, la memoria direccionable por un proceso se clasifica normalmente en cinco categorías:

- Región de código. Debe poderse leer pero puede no ser accesible para escritura.
- Región de datos estáticos de solo lectura. Suele contener variables globales constantes.
- Región de datos estáticos de lectura y escritura. Suele contener variables globales.
- El montículo o *heap*.
- La pila o *stack*.

La pila es la única región que debe ser continua, las demás regiones pueden estar fragmentadas. El *heap* debe ser gestionado por el propio proceso (por ejemplo, como en C, a través de llamadas a *malloc* y *free*), y suele utilizarse para la creación dinámica de objetos.

Dichas regiones serán determinadas por el fichero de enlazado, también llamado *linker script file*, que adjuntaremos en cada práctica. Este fichero de enlazado lo identificaremos con una terminación *.ld*.

3 Acceso al sistema de E/S

Aunque hasta ahora hemos hablado del ARM, que es el microcontrolador, el objetivo será manejar los dispositivos presentes en la placa S3CEV40. En este caso la E/S y la memoria comparten el espacio de direcciones, por lo que para leer y/o escribir en los dispositivos de E/S habrá que leer y/o escribir en *posiciones de memoria*.

A continuación se incluye la información más relevante de cada dispositivo utilizado:

- LEDs:

  - Registro de control de 11 bits (PCONB) en dirección 0x01d20008.
  - Registro de datos de 11 bits (PDATB) en dirección 0x01D2000C.
  - Configuración: pines 9 y 10 de PCONB como salida escribiendo 0.
  - Encendido/apagado: escribir 0 para encender (1 para apagar) en el bit 9 (led1) y/o 10 (led2) de PDATB.

- Pulsadores (botones):
  - Registro de control de 16 bits (PCONG) en dirección 0x01D20040. Este registro está organizado en 8 pares de bits, cada uno de los cuales controla un pin. De esta forma, por ejemplo, para acceder al pin 7 deberíamos escribir en los bits 14 y 15.
  - Registro de control de 8 bits PUPG en dirección 0x01D20048.
  - Registro de datos de 8 bits (PDATG) en dirección 0x01D20044.
  - Configuración. Pines 6 y 7 como entrada (escribir 0's en PCONG) y poner a 0 PUPG.
  - Detección de pulsación: leer de PDATG y comprobar si se está pulsando el botón. Si el bit 6 es 0, el botón 1 está pulsado (resp. bit 7 y botón 2).

- Display de 8 segmentos:
  - Registro de control: no existe (siempre configurados como salida).
  - Registro de datos: escribiremos en la dirección 0x2140000 (LED8ADDR).
  - Encendido/apagado de cada segmento: cada segmento tiene asociado un bit del registro de datos. Si un bit está a 0, el segmento correspondiente está encendido. El formato de dichos bits es *a b c dp d e f g*, por lo que si queremos escribir un 1, por ejemplo, escribiremos 0x9F (10011111 en binario) en LED8ADDR.

3.1 Acceso a nivel de bit

Tal y como puede observarse en el apartado anterior, cada dispositivo de E/S tiene asociados unos registros de control y de datos. Dichos registros no son más que una dirección de memoria. Para no tener que recordar todas estas direcciones, emplearemos los mnemotécticos que aparecen entre paréntesis en el apartado anterior. Desde el punto de vista de la programación en C podemos entender dichos mnemotécnicos como una variable. De hecho, estarán definidos en uno de los ficheros del proyecto, como explicaremos más adelante. Así pues, para modificar los registros de datos y/o control bastará con leer y escribir sobre una variable.

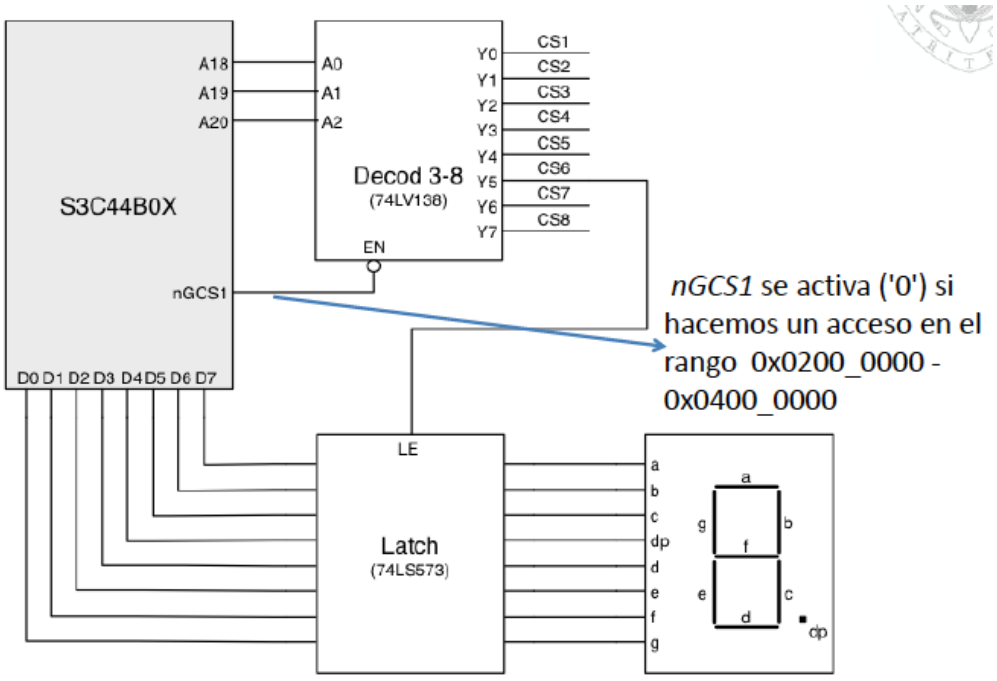


Figura 3: Display 8 segmentos

Cuando tratemos con los dispositivos de la placa, muchas veces tan solo será necesario modificar un bit de los registros. Esto es muy importante porque puede haber registros que controlen varios dispositivos, y las acciones que hagamos sobre un dispositivo no deben modificar las líneas de control y datos de otro. Para escribir sobre un subconjunto de bits utilizaremos máscaras y operaciones lógicas.

Por ejemplo, supongamos que queremos comprobar que uno de los botones está pulsado. Para leer el estado del bit 6 del PDATG bastará con hacer la AND, situando un 1 en la posición 6 de la máscara y en el resto 0's.

Código 1: Ejemplo de máscaras

```
int aux = rPDATG & 0x00000040;
if (aux==0) //Pulsado

...

rPDATB = rPDATB | 0x00000200;
```

Análogamente, para escribir un 1 y apagar el LED correspondiente al bit 9 de PDATG, bastaría con hacer la OR entre el contenido de PDATG y una máscara con un 1 en el bit 9

y en el resto todo 0's.

Nota: ha de recordarse que el bit menos significativo es el bit 0.

4 Trabajando con interrupciones en la placa S3CEV40

A la hora de interaccionar con los distintos elementos de E/S podemos utilizar dos métodos:

1) E/S programada. Con este procedimiento, el procesador *pregunta* continuamente al dispositivo de E/S si tiene algún dato nuevo, deteniendo así la ejecución del programa en curso. Por tanto es altamente ineficiente.

2) E/S por interrupciones. Con este segundo método, el procesador continúa su ejecución y es el dispositivo de E/S quien le *avisa* de que tiene un dato disponible, por lo que no detiene la ejecución del programa en curso. En nuestro caso, usaremos el sistema de E/S por interrupciones de la placa S3CEV40, salvo que se especifique lo contrario. De hecho, en un SED los elementos se comunicarán por medio de mensajes que funcionarán como si de una interrupción se tratase.

El esquema básico de interconexiones del sistema de interrupciones de la placa S3CEV40 puede observarse en la figura 4. El procesador ARM7TDMI dispone de tres líneas que permiten interrumpir su ejecución de manera externa: una línea de RESET de comportamiento asíncrono, que permite realizar un reinicialización de sus componentes y genera una excepción de Reset, y dos líneas, IRQ y FIQ de comportamiento síncrono y enmascarables, que permiten a los dispositivos de E/S solicitar la atención del procesador y generar interrupciones IRQ y FIQ respectivamente.

Las interrupciones IRQ y FIQ tienen por propósito ejecutar el código necesario para atender al dispositivo que ha solicitado la interrupción. Las líneas IRQ y FIQ pueden enmascararse mediante dos bits del registro CPSR, I y F respectivamente (ver figura 1). Cuando estos bits toman el valor '1' las solicitudes de interrupción efectuadas por línea correspondiente no son atendidas por el procesador. Durante la inicialización del sistema es crucial enmascarar ambas líneas ya que éste no se encuentra aún preparado para que las interrupciones sean atendidas.

Desde el punto de vista del procesador, todas las excepciones, incluyendo IRQ y FIQ, son autovectorizadas. Esto quiere decir que el procesador genera automáticamente para cada tipo de excepción un vector (dirección), que es cargado directamente en el contador de programa. En esta dirección de memoria debe almacenarse una instrucción de salto a la rutina de tratamiento correspondiente a la excepción.

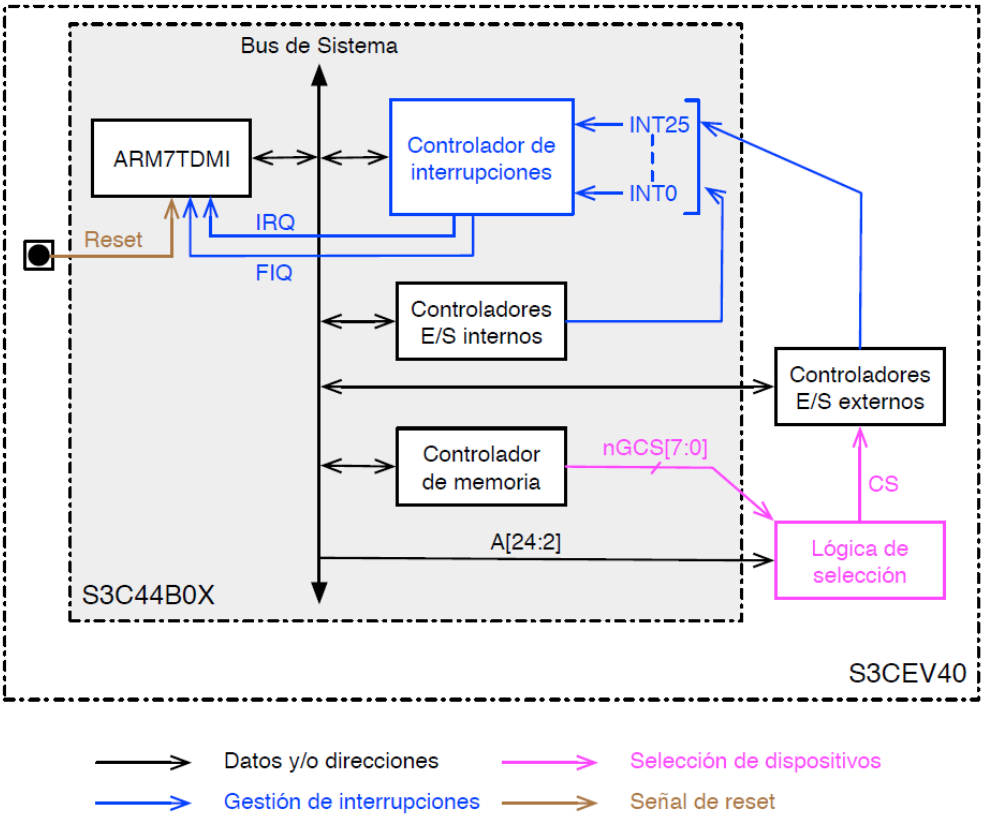


Figura 4: Sistema de interrupciones de la placa S3CEV40

Si varios dispositivos comparten la línea de petición de interrupción, ya sea IRQ o FIQ, generarán el mismo tipo de interrupción y ejecutarán la misma rutina de tratamiento. En este caso, por lo tanto, es necesario que esta rutina identifique cuál ha sido el dispositivo que ha generado la interrupción. Para hacerlo, debe leer los registros de estado de los controladores de E/S asociados para comprobar si tienen una interrupción pendiente.

La figura 5 muestra la descripción de las 26 líneas de interrupción gestionadas por el controlador de interrupciones. Estas líneas permiten gestionar 30 fuentes de interrupción distintas. La línea 21 es compartida por 4 fuentes de interrupción y la línea 14 por 2, el resto de líneas tienen asociada una única fuente de interrupción.

4.1 Configuración del controlador de interrupciones

Además de conocer las diversas líneas de interrupción, es necesario configurar correctamente el controlador de la placa. Para ello utilizaremos los siguientes registros:

- INTCON (*Interrupt Control Register*). Registro de cuatro bits en el que sólo se utilizan tres de ellos:

Nº/Bit	Nombre	Fuente	Vector
25	EINT0	Interrupción externa 0	0x20
24	EINT1	Interrupción externa 1	0x24
23	EINT2	Interrupción externa 2	0x28
22	EINT3	Interrupción externa 3	0x2c
21	EINT4/5/6/7	Interrupciones externas 4, 5, 6 y 7	0x30
20	TICK	Interrupción de *tick* del RTC	0x34
19	ZDMA0	Interrupción del ZDMA0	0x40
18	ZDMA1	Interrupción del ZDMA1	0x44
17	BDMA0	Interrupción del BDMA0	0x48
16	BDMA1	Interrupción del BDMA1	0x4c
15	WDT	Interrupción del Watch-Dog Timer	0x50
14	UERR0/1	Interrupciones de error de las UART0/1	0x54
13	TIMER0	Interrupción del Timer0	0x60
12	TIMER1	Interrupción del Timer1	0x64
11	TIMER2	Interrupción del Timer2	0x68
10	TIMER3	Interrupción del Timer3	0x6c
9	TIMER4	Interrupción del Timer4	0x70
8	TIMER5	Interrupción del Timer5	0x74
7	URXD0	Interrupción de recepción de la UART0	0x80
6	URXD1	Interrupción de recepción de la UART1	0x84
5	IIC	Interrupción de controlador de bus IIC	0x88
4	SIO	Interrupción del controlador SIO	0x8c
3	UTXD0	Interrupción de envío de la UART0	0x90
2	UTXD1	Interrupción de envío de la UART1	0x94
1	RTC	Interrupción de alarma del RTC	0xa0
0	ADC	Interrupción *EOC* del conversor ADC	0xc0

Figura 5: Líneas de interrupción

  - V (bit [2]) = 0, habilita el Modo Vectorizado.
  - I (bit [1]) = 0, habilita la línea IRQ.
  - F (bit [0]) = 0, habilita la línea FIQ.

- INTMOD (*Interrupt Mode Register*). Registro con un bit por línea: a '0' para que active IRQ o a '1' para que active FIQ.

- INTPND (*Interrupt Pending Register*). Registro con un bit por línea: a '0' si no hay solicitud y a '1' si hay una solicitud. Este registro se actualiza incluso cuando la línea está enmascarada y por lo tanto no se traslada la interrupción al procesador.

- INTMSK (*Interrupt Mask Register*). Registro con 28 bits. El bit 27 está reservado. El bit 26 permite enmascarar todas las líneas (máscara global). El resto de los bits, uno por línea, cuando toman valor '0' habilitan la interrupción de la línea correspondiente y cuando toman valor '1' la enmascaran.

- I_ISPR (*IRQ Interrupt Service Pending Register*). Registro con un bit por línea. Indica la interrupción que se está sirviendo actualmente. Aunque haya varias peticiones pendientes, cada una con su correspondiente bit del registro INTPND activo, sólo uno de los bits del registro I_ISPR estará activo (el más prioritario).

- I_ISPC (*IRQ Interrupt Service Pending Clear* register). Registro con un bit por línea. Permite borrar el bit correspondiente del INTPND escribiendo '1' en la posición correspondiente. Si lo que se escribe es un '0' el bit correspondiente de INTPND permanece inalterado. Es preciso resaltar que mediante la escritura en este registro se indica al controlador de interrupciones que ha finalizado la rutina de servicio y, que por lo tanto, puede comenzar a atender una nueva solicitud. **Importante: en este registro tan solo se puede escribir. En otras palabras, escribiremos un '1' en aquellas líneas cuya interrupción haya sido atendida.**

- F_ISPC (*FIQ Interrupt Service Pending Clear* register). Igual que I_ISPC pero para la línea FIQ.

Por último, si utilizamos hardware externo a la placa S3C44B0X (pero dentro de la S3CEV40, véase la figura 4) habrá que tener en cuenta los controladores E/S externos.

- EXTINT. Es un registro de 32-bits donde cada grupo de 3 bits sirve para configurar si la correspondiente línea externa de interrupción *EINTi* (EINT0-EINT7) será activa por nivel o por flanco. No todos los bits de este registro se utilizan para dicha configuración. Consúltese el manual [2].

- EXTINTPND. Es un registro de interrupciones externas pendientes de tan solo 4 posiciones. Dicho registro se utiliza para trabajar con la línea EINT4/5/6/7, que es la OR lógica de las correspondientes líneas EINT. De esta forma, cuando EINT4 esté activa, leeremos un '1' en la posición 0 del EXTINTPND. Análogamente con EINT5, EINT6, EINT7 y las posiciones 1, 2 y 3 del EXTINTPND. Para indicar que las interrupciones han sido atendidas, borraremos la línea de la correspondiente interrupción. **Importante: Para borrar una línea del EXTINTPND habrá que escribir un '1'**. Además, dado que estamos atendiendo una interrupción externa, habría que escribir un '1' en la posición 21 el I_ISPC, para marcar como atendida la línea EINT4/5/6/7 del INTPND. **Es muy importante hacerlo en este orden, primero el EXTINTPND y luego el I_ISPC**. De lo contrario atenderemos la interrupción con el I_ISPC y se generará otra nueva interrupción porque el EXTINTPND sigue activo.

Debe tenerse en cuenta que estos registros o mnemotécnicos en realidad son posiciones de memoria, como muestra la figura 6. Para más información sobre estos registros es preciso consultar el manual de referencia del S3C44B0X[2].

Registro	Dirección	R/W	Valor de Reset
INTCON	0X01E00000	R/W	0x7
INTPND	0X01E00004	R	0x00000000
INTMOD	0X01E00008	R/W	0x00000000
INTMSK	0X01E0000C	R/W	0x07FFFFFF
I_ISPR	0X01E00020	R	0x00000000
I_ISPC	0X01E00024	W	Undef
F_ISPC	0X01E0003C	W	Undef

Figura 6: Registros de interrupción

5 Estructura de un programa

En esta sección describiremos la estructura general de un programa. Aunque en este curso trabajaremos fundamentalmente programando en C, será necesario aprender a combinarlo con código ensamblador.

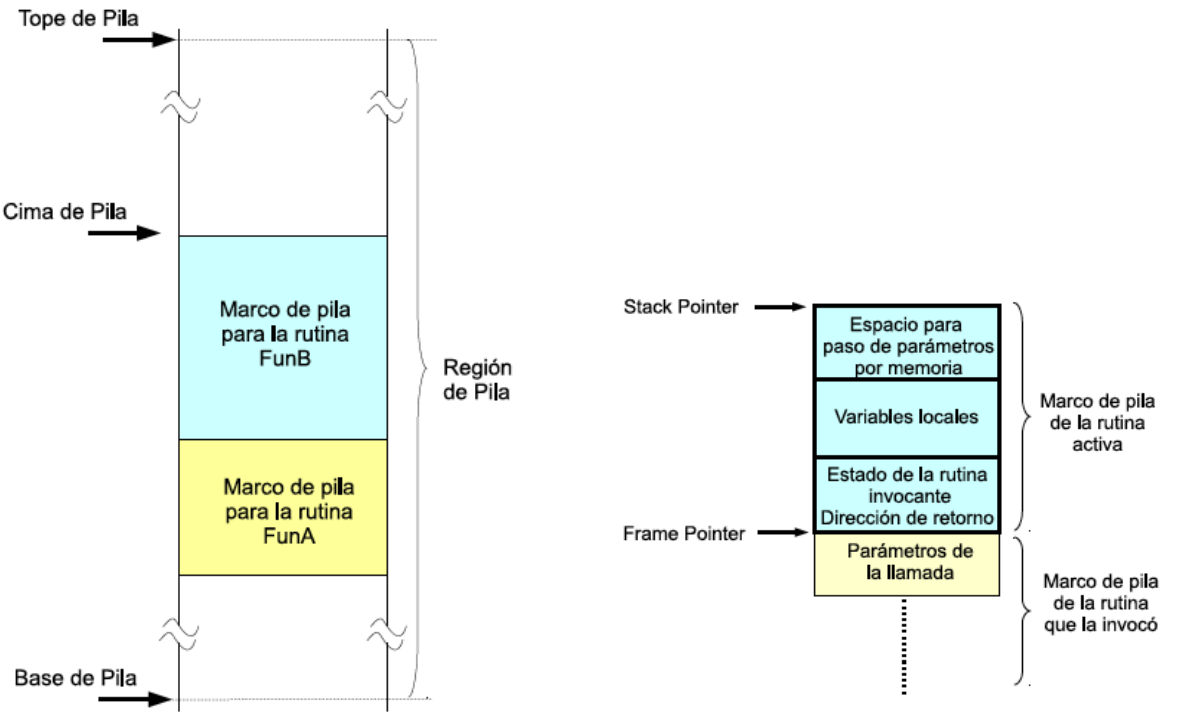
5.1 Llamadas a subrutina

Las subrutinas (funciones o procedimientos) son abstracciones que se usan en los lenguajes de alto nivel para simplificar el código y poder reusarlo. A la subrutina se la llama (invoca) desde el programa invocante mediante una instrucción particular. Pero antes de llamar a la subrutina tenemos que haber colocado los datos de entrada a ésta en un lugar accesible (registros reservados o pila). En general al trabajar con subrutinas se divide el trabajo a realizar entre el programa invocante y la subrutina:

- *Programa invocante*: Poner los parámetros o argumentos de entrada a la subrutina en un lugar donde sean accesibles por ésta.
- *Programa invocante*: Transferir el control a la subrutina. Tenemos dos opciones, igualmente válidas

  1. `BL Etiqueta  @LR=PC+4, PC=posición de Etiqueta`
  2.
```
MOV LR,PC      @LR=PC+8
LDR PC,=FUNC   @La dirección FUNC se resuelve al enlazar
Instrucción    @Esta es la dirección de retorno
```

- *Subrutina*: Ejecutar la tarea deseada usando los recursos necesarios. Nótese que antes de ejecutar la tarea, es posible que la subrutina deba salvar ciertos registros en la pila.
- *Subrutina*: Poner el resultado en un lugar accesible al programa. En el ARM la subrutina devuelve un único parámetro de salida con el resultado. En nuestro caso, vamos a trabajar siempre con datos de tamaño menor o igual a 4 bytes, con lo que



(a) Pila de llamadas

(b) Marco de activación

Figura 7: Pila de llamadas. (a) Ejemplo del estado de la pila de llamadas en el caso de que una función FunA haya invocado la rutina FunB y esta última se encuentre en ejecución.(b) Estructura del marco de activación de FunB cuando está en ejecución.

el resultado siempre se devolverá únicamente en R0. Por el contrario, si el resultado ocupase más de 4 bytes, se usarían también los otros registros del R1 al R3.

- *Subrutina*: Devolver el control al punto inicial, manteniendo el contexto. Análogamente a la llamada, disponemos de dos métodos para devolver el control a la subrutina invocante:

  1. `BX LR`
  2. `MOV PC,LR`

La figura 7 muestra la evolución de la pila de llamadas y del marco de activación para un hipotético ejemplo en el que FunA llama a la rutina FunB. Obsérvese que la parte amarilla se corresponde con los datos del programa llamante o invocador (paso 1), mientras que la parte azul lo hace con los datos del programa invocado (comienzo del paso 3).

El paso de parámetros sigue el estándar ARM Architecture Procedure Call Stack (*AAPCS*). Dicho estándar especifica una serie de normas para que las subrutinas puedan ser compiladas o ensambladas por separado y que, a pesar de ello, puedan interactuar entre ellas. Dado

que el objetivo del presente curso es programar en C, y el compilador respetará el estándar por defecto, no profundizaremos más en el AAPCS. De cualquier forma, el alumno puede obtener más información en [1].

5.2 Inicializando un programa

Un programa C siempre ha de tener una función *main* por la que debe empezar la ejecución del mismo. Esta función no es diferente de ninguna otra función, en el sentido de que construirá en la pila su marco de activación y al terminar lo deshará y retornará, copiando en PC lo que tuviese el registro LR al comienzo de la propia función *main*.

En nuestro contexto, donde no hay sistema operativo y cargamos directamente el programa en memoria para su ejecución, hay dos motivos por los que no podemos iniciar el programa en la función *main*:

- La pila no estaría inicializada, es decir, habría basura en el registro SP, y
- No se habría pasado una dirección de retorno, es decir, habría basura en LR.

Código 2: Ejemplo de inicialización en ensamblador

```
.extern main          @Simbolo global main, definido en otro fichero
.global start         @Simbolo global start, comienzo del programa
.equ STACK, 0x0c7ff000  @Cte para la direccion de inicio de SP
.text                 @Comienzo de la region de instrucciones
start:
  ldr sp,=STACK       @Inicializamos SP con la macro LDR
  mov fp,#0           @Inicializamos FP a 0
  mov lr,pc           @Salvamos el PC+4
  ldr pc,=main        @Saltamos a main
End:
  B End               @Salto incondicional a End
.end                  @Terminador .end

@Cualquier cosa posterior al .end sera ignorada por el compilador
```

El segundo motivo lo podríamos solventar haciendo que *main* no terminase (esperase en un bucle infinito), pero seguiríamos teniendo el problema de la inicialización de la pila. Por ello, siempre que hagamos un proyecto en C añadiremos un fichero ensamblador con la rutina de inicio. Esta rutina se encargará de preparar el sistema, luego invocará la función *main* y retomará el control cuando termine dicha función. Finalmente esperará en un bucle infinito.

El cuadro 2 nos muestra un posible código para esta rutina de inicialización.

Como vemos, además de lo explicado anteriormente la rutina presentada en el cuadro 2 pone a 0 el registro FP antes de llamar a la función *main*. Esto se hace así para dejar una señal en el marco de activación de *main* que indique que es el primer marco de la pila. Esto lo necesita el depurador cuando le pedimos hacer una traza de la pila de llamadas (*call trace*).

5.2.1 Programa con E/S programada

Si el programa funciona con E/S programada, la estructura del método *main* será la mostrada por el cuadro 3.

Código 3: Estructura del método *main* con E/S programada

```
int main(){
  //Configurar dispositivos como e/s
  //Inicializar dispositivos
  while(true){
      //Lectura disp. de entrada
      //Calcular nuevas salidas
      //Escribir disp. de salida
  }
}
```

Dado que la E/S es por espera activa, **la lectura de los datos de entrada contendrá a su vez bucles esperando hasta que haya algún dato**.

5.2.2 Programa con E/S por interrupciones

Si el programa utiliza el sistema de interrupciones de la placa S3CEV40 la estructura del método *main* será la mostrada por el cuadro 4.

Código 4: Estructura del método *main* con E/S por interrupciones

```
int main(){
  //Configurar dispositivos como e/s
  //Inicializar dispositivos
  while(true){
      //NADA
  }
}
```

Dado que los dispositivos utilizarán las interrupciones, el bucle del programa principal es un loop vacío. Cuando se produzca una interrupción, el programa principal saltará a la rutina de tratamiento correspondiente, atenderá la interrupción y por último, el dispositivo de E/S devolverá el control al programa principal, es decir, al *while(true)*.

Ha de notarse que en este caso las rutinas de tratamiento de interrupción deben ser asignadas durante la configuración de los dispositivos.

5.3 La rutina de tratamiento de interrupción

En la segunda parte de la práctica (y en el resto de prácticas del curso) debemos trabajar con interrupciones. Para ello hemos de efectuar una configuración correcta.

Uno de los puntos críticos consiste en decirle al microcontrolador qué función debe ejecutarse cuando se produzca la interrupción que estamos esperando. Es decir, hay que establecer la rutina de tratamiento de dicha interrupción. El cuadro 5 contiene un ejemplo sobre cómo realizar el establecimiento para el caso del *timer0*, uno de los periféricos que se utilizarán durante el curso.

Cada interrupción tiene una variable asociada *pISR_interrupcion* (definidas en el fichero auxiliar *44b.h*). En el caso del *timer0*, la variable se denomina *pISR_timer0*. Para efectuar el establecimiento, hay que asignarle a esta variable la dirección de comienzo de nuestra rutina, que en el ejemplo se llama *timer0_rti*.

Código 5: Establecimiento de la rutina de tratamiento de la interrupción

```
void init_timer0(){
//Inicializar timer0
pISR_TIMER0 = (unsigned)timer0_rti;
...
}

void timer0_rti(){
//Rutina de tratamiento del timer0
...
}
```

5.4 Consejos de programación

Dado que hemos de utilizar máscaras y trabajar a nivel de bit sobre cada línea de interrupción, es recomendable utilizar una serie de constantes definidas en el fichero auxiliar

44b.h, para evitar posibles errores de programación. Además de las variables *pISR*, existe una serie de constantes que siguen la nomenclatura *BIT_interrupcion*. Cada una de estas constantes equivale a un 0x1 desplazado tantas posiciones como el lugar que ocupa la línea de interrupción correspondiente.

En el ejemplo del cuadro 6 se muestra la definición de la constante *BIT_TIMER0*, cuyo valor es un 1 desplazado 13 posiciones, ya que el *timer0* se controla con la línea de interrupción 13. A continuación podemos observar cómo se desenmascararía el *timer0*, y cómo se borraría su interrupción del registro de pendientes. De otra manera, deberíamos introducir manualmente el valor 0x2000, lo cual es más propenso a errores.

Por último, obsérvese que el proceso de borrado de la interrupción consiste en escribir un 1 en la posición correspondiente del registro I_ISPC. Este registro solo tiene permisos de escritura, por lo que **NO** puede leerse. Es decir, efectuar una or o una and lógica sobre el I_ISPC es incorrecto.

Código 6: Uso de macros definidas en el fichero 44b.h

```
#define BIT_TIMER0  (0x1<<13)
...
rINTMSK = rINTMSK & ~BIT_TIMER0;
...
rI_ISPC = BIT_TIMER0;
...
```

6 Desarrollo de la práctica

La práctica está dividida en dos partes. Una primera parte guiada, en la que nos familiarizaremos con el entorno de desarrollo. Y una segunda parte en la que el alumno tendrá que realizar ciertos ejercicios.

6.1 Parte guiada

En esta primera parte, seguiremos los siguientes pasos:

1) Abrir el Eclipse y crear un proyecto C nuevo tal y como muestra la figura 8. En *Project type* seleccionamos *ARM Cross Target Application* y *Empty Project*. En *Toolchains* seleccionamos *ARM Windows GCC (Sourcery G++ Lite)*. Finalmente pulsamos *Next* y *Finish*.

2) Descargamos la carpeta *commonEclipse* del Campus Virtual y la añadimos al proyecto.

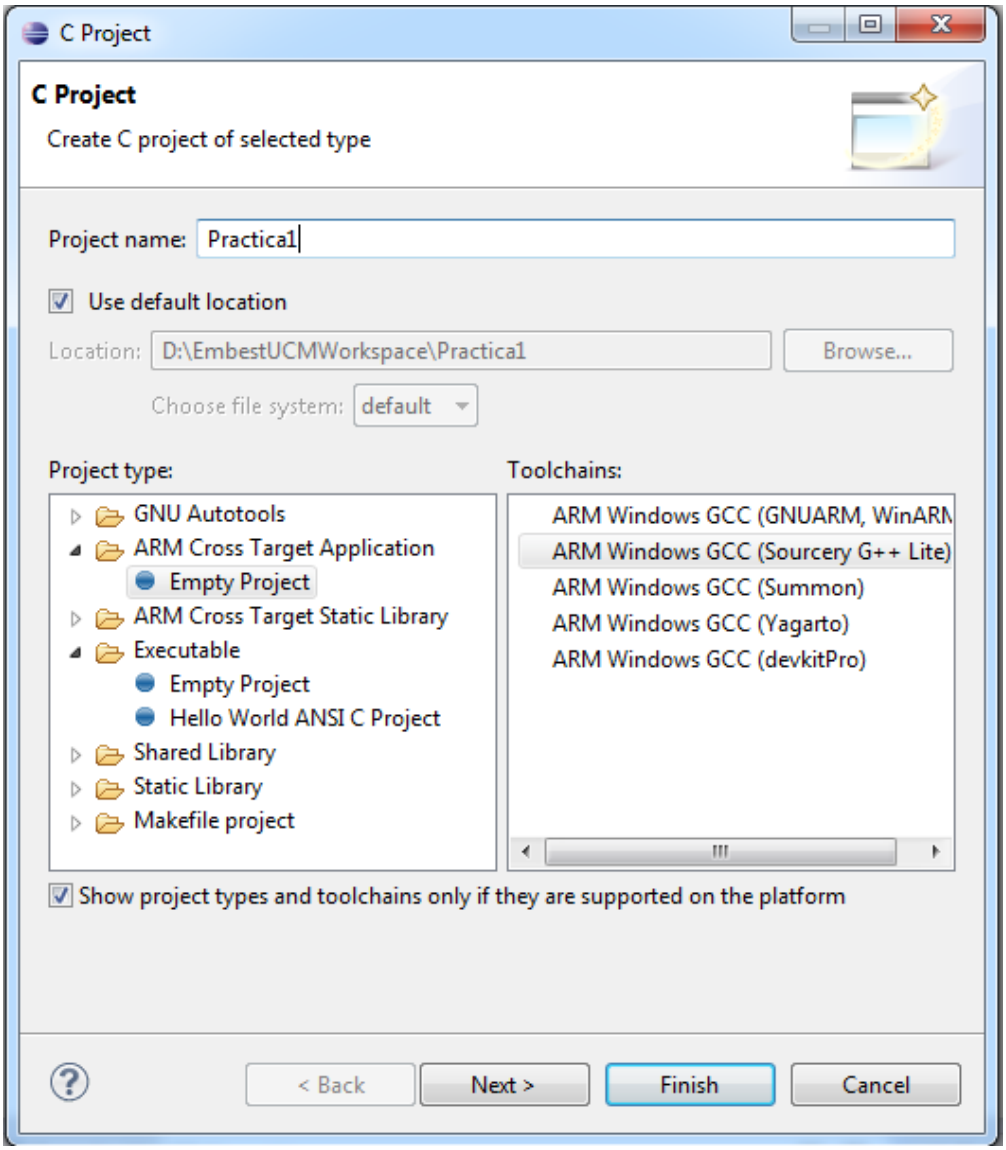


Figura 8: Creación de un proyecto C

3) Descargamos el fichero *pr1Skeleton.zip* del Campus Virtual, lo descomprimimos y copiamos los ficheros *init.asm*, *main.c* y *leds.c* carpeta del proyecto. A continuacion refrescamos el explorador del proyecto (ventana *Project Explorer*) y veremos dichos ficheros.

4) Seleccionaremos el *ram_ice.ld*, que está dentro de *commonEclipse* como fichero de enlazado. Dicho código es el que utilizará el linker para determinar las secciones del ejecutable así como su ubicación. Para ello iremos a las *Properties* del proyecto y en *C/C++ Build* → *Settings* → *Linker* → *General* seleccionamos el *ram_ice.ld*

5) Todavía dentro de *Settings*, anãdiremos la carpeta *commonEclipse* en la opción Directories del *Compiler* y del *Assembler*.

6) Compilamos el proyecto con *Build All* o con *Build Project*.

7) Si todo ha ido bien, el proyecto debería estar compilado correctamente. Ahora vamos a crear un perfil de depuración. En *Run* entramos en *Debug Configurations*, seleccionamos *GDB Hardware Debugging* y pulsamos el icono de *New Launch Configuration*, situado a la izquierda de la ventana. Damos un nombre a la configuración y seleccionamos el proyecto actual, si no lo estuviera ya.

8) A continuación configuraremos las settings para ejecutar el fichero .elf en la placa. Vamos a la pestaña *Debugger* y en *GDB Command* escribimos *arm-none-eabi-gdb*. En *Port Number* damos el valor 3333, tal y como muestra la figura 9. En la parte inferior pulsamos *Select other* y en la nueva ventana marcamos *Use configuration specific settings* y seleccionamos el *Standard GDB Hardware Debugging Launcher*.

9) Vamos ahora a la pestaña *Startup*. Desmarcamos las opciones *Reset and Delay* y *Halt*. Por último escribimos *remote reset init* en el campo de texto para comandos, situado justo debajo de *Halt*, y pulsamos *Apply* y *Close*.

10) Finalmente, ejecutaremos el binario. En primer lugar, debemos conectar el ARM con el Olimex. Para ello, lanzamos el OpenOCD. Si el OpenOCD no estuviera configurado, seguimos los pasos del videotutorial 1 (minuto 3 aproximadamente) para configurarlo. Si está configurado, nos vamos a *External Tools Configurations*, dentro del menú *Run* y seleccionamos *OpenOCD*. Pulsamos *Run* para lanzar el OpenOCD. **Importante: la placa debe estar encendida antes de lanzar el OpenOCD.**

11) En segundo lugar, iniciamos la depuración. Vamos a *Debug Configurations*, seleccionamos el perfil de configuración que creamos en los apartados 6-8, y pulsamos *Debug*.

Si todo ha ido bien, los leds deberían conmutar cada 1s. Además, en el Eclipse la *perspectiva* habrá cambiado de C a Debug. Con esta nueva perspectiva, seremos capaces de observar distintos elementos del programa y de la placa, como por ejemplo:

1) Variables, breakpoints y registros del procesador.

2) Desensamblado del código fuente y el mismo código fuente. Esto será muy útil para depurar paso a paso con las opciones *Step into*, *Step over*, etc.

3) Contenido de la memoria. Si no está añadido ya, iremos a la pestaña *Memory*, situada en la parte inferior de la perspectiva de depuración, y pulsaremos *Add Memory Monitor* (signo +), introduciendo la dirección a partir de la cual queremos observar la memoria. **Importante: recordad que los datos vienen en Little Endian**.

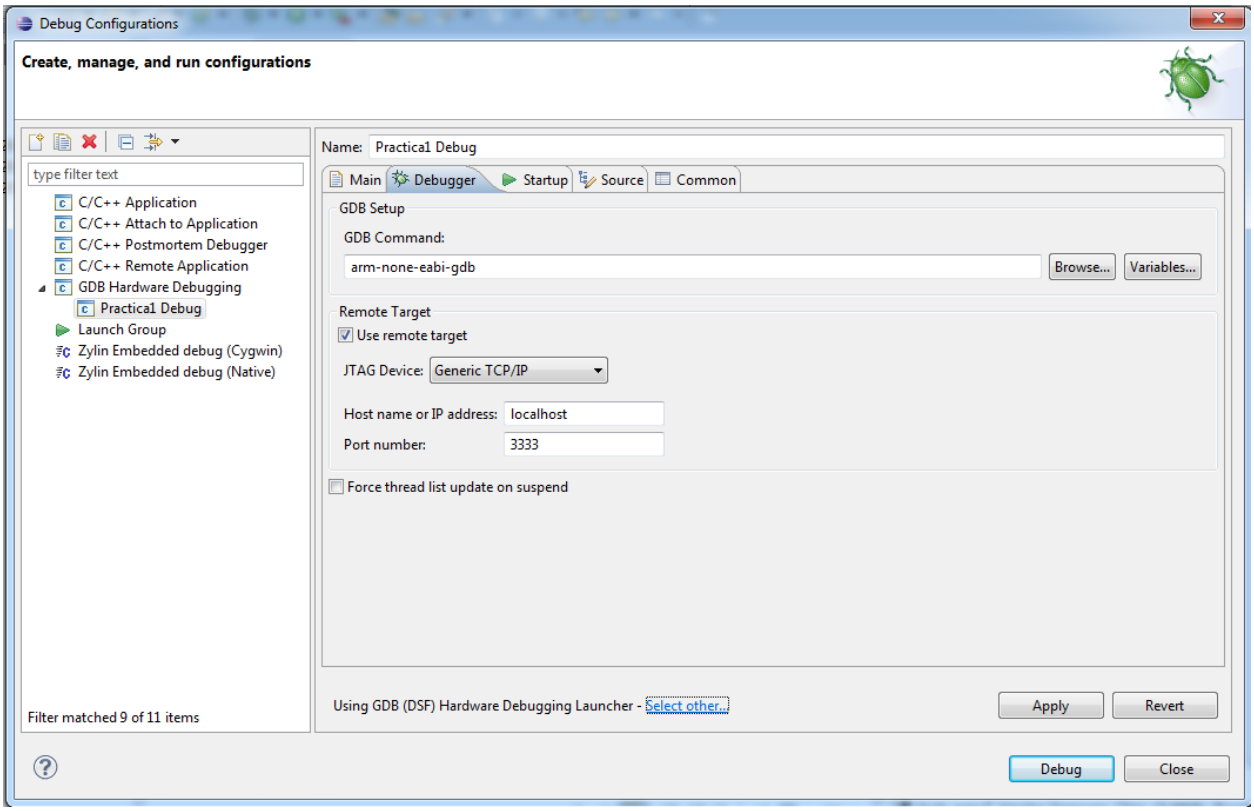


Figura 9: Configuración del depurador

4) Registros del controlador. Con este visor podremos observar los registros de configuración y datos de los distintos dispositivos de E/S, los registros de interrupción, etc. Si no está añadido, basta con ir a *Window* → *Show view* → *Other* y seleccionar *EmbSys Registers*, dentro de la carpeta *Debug*.

En resumen, la perspectiva de depuración debería mostrar un aspecto similar al de la figura 10.
Para terminar la depuración, basta con pulsar con el botón derecho del ratón sobre los procesos lanzados (el proyecto y el OpenOCD) y seleccionar *Terminate and Remove*. Otra opción consistiría en pulsar el botón de parada, que aparece en la barra de herramientas.

6.2 Parte no guiada

En esta segunda parte trabajaremos también con pulsadores y con el display 7 segmentos, utilizando los registros de configuración explicados en la sección 3.

6.2.1 E/S programada

En primer lugar, se propone modificar el fichero *main.c* para que los leds conmuten cada vez que se pulse cualquiera de los botones (o ambos). Para ello se utilizará el método de la E/S

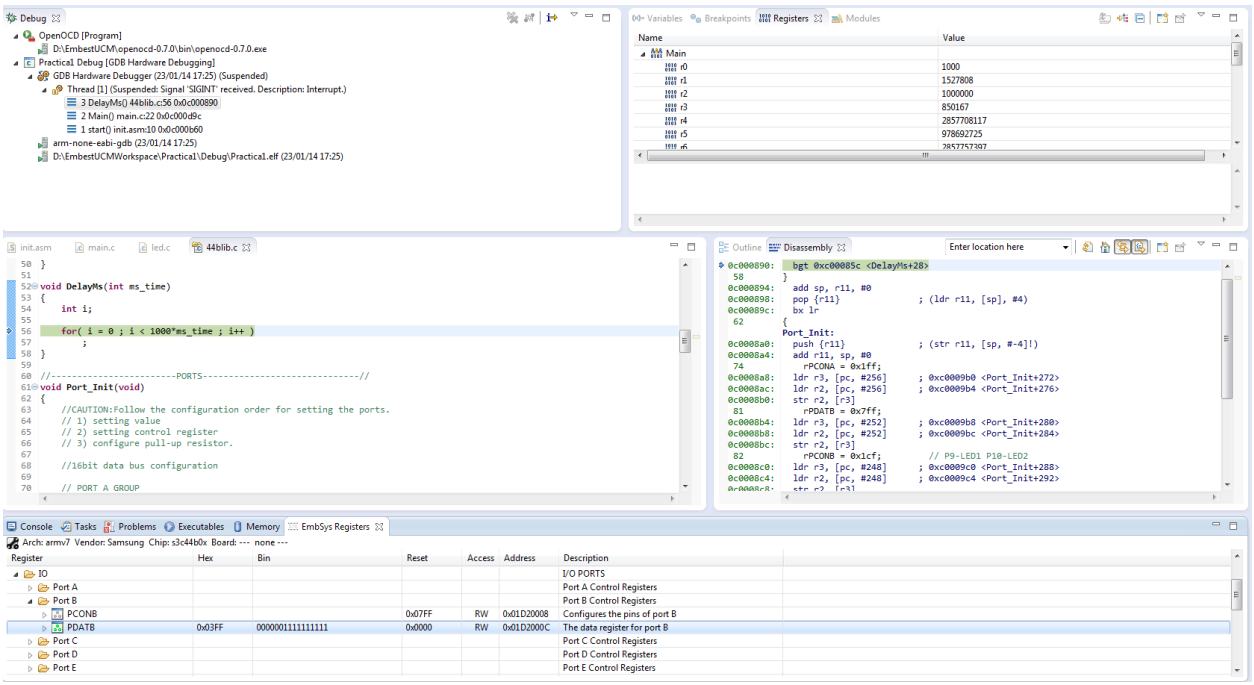


Figura 10: Perspectiva de depuración

programada.

Sugerencia: Inicializar los botones como entrada con un procedimiento button_init(), e incluir un bucle de espera, dentro del bucle principal, que estudie continuamente el valor de rPDATG.

Nota: Llamar a la función DelayMs(100) después de leer rPDATG y después de hacer el led_switch().

6.2.2 E/S por interrupciones

A continuación realizaremos el mismo programa pero usando interrupciones. Para ello añadiremos el fichero *button.c* y completaremos la rutina de inicialización de los botones (*Eint4567_init*) y la rutina de tratamiento interrupciones (*Eint4567_ISR*). Además habrá que invocar la rutina de inicialización de los botones en el *main* y modificar el bucle principal de acuerdo a lo explicado en la sección 5.2.2.

6.2.3 Leds, botones y 8-segmentos

Finalmente practicaremos con el display 8-segmentos. Para ello, agregaremos el fichero *led8.c* al proyecto. Este fichero contiene las rutinas necesarias para escribir en el 8-segmentos. Como puede observarse en la función *D8Led_symbol*, la visualización de un dígito se realiza mediante la escritura en la variable *LED8ADDR*. En dicha variable escribiremos los 8 bits que indican qué segmentos se iluminan y cuáles no.

En este apartado se pide modificar la rutina de tratamiento de interrupción de los botones para implementar un contador módulo n ($n \leq 16$) ascendente/descendente, de tal manera que cuando se pulse el botón izquierdo la cuenta sea ascendente y cuando se pulse el derecho sea descendente. El valor del contador deberá visualizarse en el display. Por último, el led izquierdo se encenderá cuando se pulse el botón izquierdo, y análogamente el derecho cuando se pulse el botón derecho.

Bibliografía

[1] The arm architecture procedure call standard. Accesible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042d/index.html

[2] S3C44B0X RISC Microprocessor. Accesible en el Campus Virtual