Práctica 3

Mapa de memoria, modos de ejecución y gestión de excepciones

3.1. Objetivos de la práctica

En esta práctica finalizaremos el estudio del procesador ARM7TDMI analizando sus modos de ejecución, sus excepciones y su sistema de memoria. En la práctica comenzaremos a analizar algunas características del sistema en chip S3C44BOX y la placa S3CEV40 utilizados en el laboratorio. Los principales objetivos de la práctica son:

- Conocer los modos de ejecución del procesador
- Entender el sistema de tratamiento de excepciones.
- Conocer el sistema de memoria de la placa S3CEV40 y la configuración básica del procesador para su gestión.
- Aprender a diseñar un programa residente en memoria FLASH.

3.2. Modos de ejecución del ARM7TDMI y gestión de excepciones

Una excepción es un mecanismo que permite atender eventos inesperados, con origen interno (ej: intento de ejecutar una instrucción no definida) o externo (ej: solicitud de interrupción externa por parte de un dispositivo). Normalmente cuando el origen es externo se utiliza el nombre de interrupción. La idea es sencilla, cuando se produce una excepción el procesador interrumpe de forma controlada su ejecución y pasa a ejecutar una rutina específica que tratará esa excepción. Debemos tener en cuenta que pueden producirse varias excepciones simultáneamente, por lo que debe establecerse una serie de prioridades a la hora de atenderlas. El procesador ARM7TDMI dispone de varios modos de ejecución diferentes para tratar las excepciones.

3.2.1. Modos de Ejecución

Hasta ahora todos los programas que hemos utilizado en las prácticas ejecutaban en un sólo modo de ejecución, de hecho no nos hemos preocupado del modo en que se ejecutaban. Sin embargo los procesadores de ARM cuentan con varios modos de ejecución que permiten entre otras cosas la gestión eficiente de excepciones.

El ARM7TDMI dispone de 7 modos de ejecución diferentes, descritos en la tabla 3.1. El modo de ejecución se determina por los cinco bits menos significativos del registro de estado (`M[4:0]`), cambiando estos bits el procesador cambia de modo, aunque no es la manera habitual de hacerlo . La figura 3.1 describe los distintos campos del registro de estado, la explicación de cada uno de ellos se dio en la práctica 1.

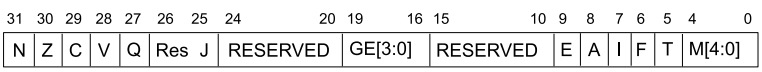

Figura 3.1: Descripción del Registro de estado (CPSR).

El modo usuario (usr) es no privilegiado y el resto son privilegiados. En un modo no privilegiado no se puede acceder a determinados recursos de la arquitectura ni escribir

Tabla 3.1: Modos del procesador

Modo del procesador	Uso
usr	Ejecución de código de usuario
fiq	Servicio de int. rápidas
irq	Servicio de int. lentas
svc	Modo protegido para sistema operativo (int. sw)
abt	Procesado de fallos de acceso a mem
und	Manejo de instrucc. indefinidas
sys	Ejecución de tareas del SO

libremente en el registro de estado, impidiendo que se pueda cambiar de modo con un mecanismo distinto a una excepción. Esto permite al sistema operativo restringir el acceso de los programas en modo usuario a los recursos arquitectónicos.

De los modos privilegiados cinco son conocidos como modos de excepción, porque están directamente relacionados con un tipo de excepción: FIQ, IRQ, Supervisor, Abort y Undefined. Estos modos tienen algunos registros propios para evitar que el procesador entre en un modo inestable cuando se produce la excepción y para reducir los accesos a la pila necesarios para salvaguardar el estado previo a la excepción.

El séptimo y último modo, System, sólo está disponible en las versiones más modernas de la arquitectura ARM. A diferencia del resto de modos privilegiados, el paso a este modo no ocurre mediante una excepción. El modo System cuenta con los mismos registros que el modo de usuario, pero sin las limitaciones del modo usuario. Lo emplea el sistema operativo cuando necesita acceder a ciertos recursos del sistema desde fuera de un modo de excepción.

Registros y modos de ejecución

Como hemos mencionado en prácticas anteriores, la arquitectura dispone de 37 registros de 32 bits, incluyendo el contador de programa. Estos registros se organizan en bancos parcialmente solapados. El modo del procesador determina qué registros son accesibles en cada momento. En cada instante podemos acceder a 15 registros de propósito general (r0-r14), el registro de estado CPRS, el registro de sombra SPRS (salvo en modo usuario) y el contador de programa PC (r15). La figura 3.2 ilustra los registros visibles en cada modo de ejecución.

Cambio de modo de ejecución

Hay dos formas de cambiar de modo de ejecución: mediante una excepción o modificando los bits `M[4:0]` del registro de estado. El primer mecanismo es el único que permite el cambio de modo cuando se está en modo usuario y generalmente no es controlado por el programador. No obstante, mediante la instrucción `swi` (interrupción software), el programador puede generar una excepción que produce el cambio a modo supervisor. Este es el mecanismo utilizado por los sistemas operativos para controlar el acceso a los recursos protegidos, y recibe el nombre de llamada al sistema. Habitualmente, esta llamada al sistema se realiza a través de una función de la biblioteca estándar de C.

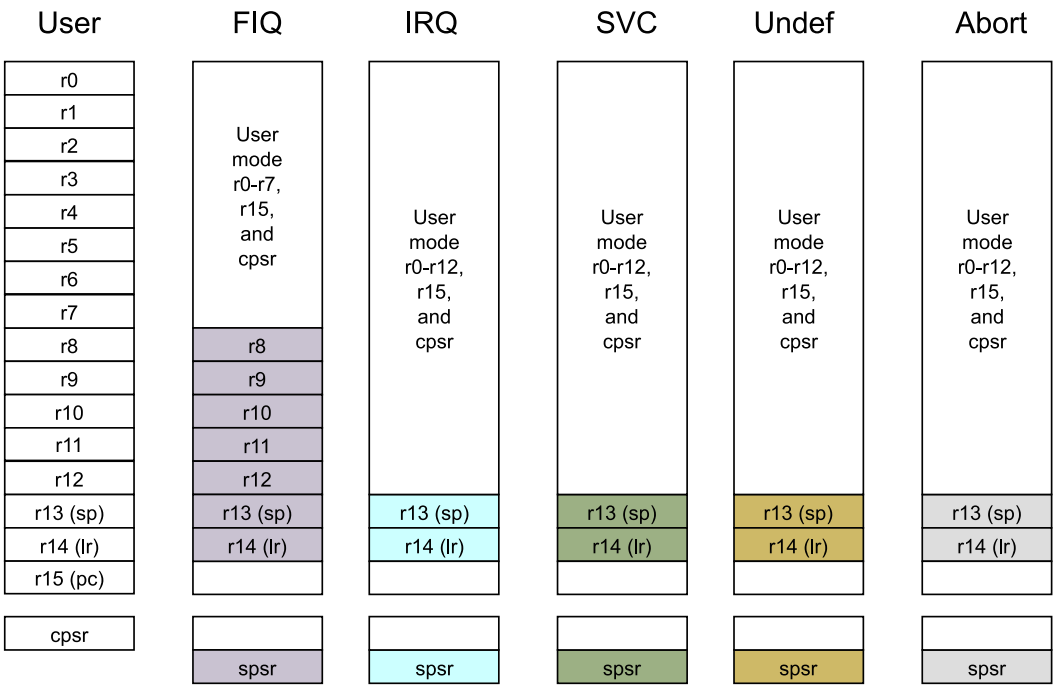

Figura 3.2: Registros visibles en cada modo de ejecución

El segundo mecanismo sólo está disponible cuando se está en un modo privilegiado. En este caso escribiendo el valor correcto en los cinco bits menos significativos del registro de estado se produce el cambio de modo. Para ello deben utilizarse las instrucciones `mrs` y `msr`.

Por ejemplo, una posible secuencia de instrucciones para pasar a modo `Undef` sería la siguiente:

```
.equ  MODEMASK,0x1f            /* Para selección de M[4:0] */
.equ    UNDEFMODE,  0x1b       /* Código de modo Undef */


mrs     r0,cpsr                /* Llevamos el registro de estado a r0 */
bic     r0,r0,#MODEMASK        /* Borramos los bits de modo de r0 */
orr     r1,r0,#UNDEFMODE       /* Añadimos el código del modo Undef y
                                  copiamos en r1 */
msr     cpsr_cxsf,r1           /* Escribimos el resultado en el registro de
                                  estado, cambiando de éste los bits de del
                                  campo de control, de extension, de estado y
                                  los de flag.  */
ldr     sp,=UndefStack         /* Una vez en modo Undef copiamos la
                                  dirección de comienzo de la pila */

                    /* Estamos en modo Undef */
```

3.2.2. Excepciones

La arquitectura ARM7TDMI reconoce, además de la excepción Reset típica de todos los procesadores, 6 excepciones adicionales. Veamos una breve descripción (para más información consultar [arm]):

Reset Se produce cuando se activa la señal externa de reset del sistema.

Undef Se produce cuando se intenta ejecutar una instrucción no definida. Si la condición de la instrucción no se cumple (recordemos que todas las instrucciones son condicionales) entonces la excepción no se produce.

SWI Se produce cuando se ejecuta la instrucción `swi` (interrupción software).

IRQ Se produce cuando se activa la línea de interrupciones externas IRQ.

FIQ Se produce cuando se activa la línea de interrupciones externas rápidas FIQ.

Abort Se distinguen dos tipos de excepción:

- **Prefetch Abort** Cuando se realiza la búsqueda (fetch) de una instrucción en una dirección no válida, pasando a modo `PAbort`. Es el controlador de memoria el responsable de generar la interrupción.
- **Data Abort** Cuando se intenta acceder a memoria en una posición no válida, para lectura o escritura de datos. Se pasa a modo `DPAbort`. Es el controlador de memoria el responsable de generar la interrupción.

La tabla 3.2, ordenada de mayor a menor prioridad, muestra la correspondencia entre las excepciones, los modos de ejecución y las direcciones de vector.

Tabla 3.2: Correspondencia entre excepciones, modos y vectores.

Prioridad	Excepción	Modo	Vector
1	Reset	SVC	`0x00`
2	Data Abort	Abort	`0x10`
3	FIQ	FIQ	`0x1C`
4	IRQ	IRQ	`0x18`
6	Prefetch Abort	Abort	`0x0C`
7	Instruccion no definida	Undef	`0x04`
8	SWI	SVC	`0x08`

Como vemos, las excepciones del ARM7TDMI son autovectorizadas. Esto quiere decir que el vector de excepción se genera automáticamente en función de la excepción. Cuando se produce una excepción el procesador realiza automáticamente los siguientes pasos:

1. Almacena la dirección de retorno en el registro r14 del modo de ejecución para el tratamiento de la excepción. En realidad el valor almacenado depende del tipo de excepción (consultar [arm]) lo que hace que el retorno de cada rutina de tratamiento de excepción sea distinto, como veremos más adelante.

```
R14_<modo_de_excepcion>  = direccion de retorno
```

2. Copia el registro de estado (CPSR) en el registro `SPSR` del modo de ejecución correspondiente a la excepción.

```
SPSR_<modo_de_excepcion> = CPSR
```

3. Pone el código del modo de ejecución correspondiente a la excepción en los bits `M[4:0]` del registro de estado.

```
CPSR[4:0] = código del modo de excepción
```

4. Cambia al estado ARM, si no lo estuviese ya.

```
CPSR[5] = 0 /* Cambiar a estado ARM */
```

5. Si el modo para el tratamiento de la excepción es Reset o FIQ, el procesador deshabilita las interrupciones rápidas.

```
if <modo_de_excepcion> == Reset or FIQ then
    CPSR[6] = 1 /* Deshabilitar interrupciones rápidas */
/* else CPSR[6] no se cambia */
```

6. Deshabilita las interrupciones normales.

```
CPSR[7] = 1 /* Deshabilitar interrupciones normales */
```

7. Copia en el PC el vector correspondiente a la interrupción.

```
PC = dirección del vector de excepción
```

Resumiendo, lo que sucede ante una interrupción es que el procesador guarda el registro de estado en el registro de sombra del modo y ejecuta la instrucción que está almacenada en memoria en la dirección indicada por el vector de la excepción (ver tabla 3.2). Esta instrucción será un salto a la rutina que deba tratar la excepción o, como veremos más adelante, a una rutina que lea de memoria el lugar donde se encuentra dicha rutina y realice el salto definitivo a ésta.

Como el registro de enlace utilizado es propio de cada modo de ejecución, no se modifica el registro r14 del modo usuario. El registro r13 (SP) también es propio de cada modo, por lo que tampoco se modifica. Es más, habitualmente cada modo hará uso de una pila distinta. La rutina de tratamiento de la excepción deberá salvar no obstante el resto del estado cuando sea necesario (i.e. cuando se modifiquen otros registros), utilizando la pila propia del modo de excepción.

Para regresar al punto donde se había interrumpido la ejecución del programa, se debe restaurar el valor del `CPSR` a partir del valor guardado en el `SPSR` y se debe copiar el valor de retorno en `PC`. Las dos acciones del retorno se pueden llevar a cabo de dos maneras distintas:

- Mediante una instrucción de procesamiento de datos con el bit `S` activo (modificación del registro de estado) y empleando como registro destino `PC`. Cuando se utiliza el PC como destino en una instrucción que modifica el registro de estado, el hardware automáticamente restaura el valor de `CPSR` a partir del valor de `SPSR`.

- Guardando los valores en la pila y leyéndolos posteriormente mediante una instrucción de load múltiple (`LDM`).

Como el valor almacenado en `r14_modo` es diferente para cada excepción, la copia del valor de retorno al `PC` debe hacerse de forma distinta para cada excepción. La tabla 3.3 muestra la instrucción de retorno más adecuada para cada tipo de excepción.

Tabla 3.3: Instrucción de retorno de excepción usual.

Excepción	**Inst. Retorno**
Reset	NA
Data Abort	`SUBS PC, R14_abt, #8`
FIQ	`SUBS PC, R14_fiq, #4`
IRQ	`SUBS PC, R14_irq, #4`
Prefetch Abort	`SUBS PC, R14_abt, #4`
Undef	`MOVS PC, R14_und`
SWI	`MOVS PC, R14_svc`

Debemos tener en cuenta que si hay que hacer alguna operación sobre la dirección de retorno guardada en r14 para realizar correctamente el retorno (ver tabla 3.3), y queremos hacer el retorno recuperando los valores salvados en la pila del modo a través de la instrucción `LDM`, entonces r14 debe modificarse antes de salvarlo en la pila.

Finalmente, debemos observar que la estructura de una rutina de tratamiento de excepciones debe ser ligeramente distinta que la de una rutina normal y distinta de una excepción a otra. Por ejemplo, para tratar IRQ se puede emplear el siguiente esquema:

```
mov   ip, sp
sub   r14, r14, #4                /* se corrige el valor de retorno */
stmfd sp!, { <registros>, r14, pc}
sub   fp, ip, #4

/* cuerpo de la rutina */

ldmdb fp!, { <registros>, pc }^   /* se restaura CPSR (^) y PC */
```

Escritura de rutinas de tratamiento de interrupciones en C

Si queremos implementar las rutinas de tratamiento de interrupciones como funciones de C, debemos informar al compilador de que la función se utilizará para el tratamiento de una determinada excepción, de forma que genere el código con la estructura adecuada. En gcc esto se consigue añadiendo a la declaración de la función una directiva `__attribute__` del siguiente modo:

```
ret_val fun_name( params ) __attribute__((interrupt ( TYPE )));
```

donde `TYPE` puede ser `IRQ`, `FIQ`, `ABORT`, `UNDEF` o `SWI`.

3.3. Sistema de E/S y mapa de memoria

El funcionamiento del sistema de memoria depende en realidad de tres componentes de nuestro kit de laboratorio, el procesador (ARM7TDMI), el sistema en chip en el que se integra (S3C44BOX) y la placa sobre la que se coloca este último (S3CEV40).

3.3.1. El ARM7TDMI

El bus de direcciones del ARM7TDI es de 32 bits, por lo que es capaz de direccionar un espacio total de 4GB de memoria. Sin embargo, la E/S está localizada en memoria. Esto significa que tanto la memoria como la E/S comparten un mismo espacio de direcciones. Por consiguiente, cuando el procesador realiza una lectura o escritura en una dirección no es capaz de discriminar si ésta se realiza sobre la memoria o sobre la E/S. Para distinguir entre una y otra se requiere una lógica externa al procesador como veremos posteriormente. Además, el programador debe ser plenamente consciente de los rangos de direcciones asignados a cada subsistema.

3.3.2. El S3C44BOX

La placa Samsung que utilizamos en el laboratorio, S3CEV40, incorpora un *System on Chip* llamado S3C44BOX. En la figura 3.3 podemos ver un esquema con la estructura interna de este chip. Como vemos, el procesador ARM7TDMI se conecta directamente a través del bus del sistema con una serie de controladores y a través de un *bridge* a otra serie de controladores secundarios. Todos estos controladores y los buses están integrados dentro del chip. En esta práctica vamos a centrarnos en analizar el subsistema de memoria, gobernado por el controlador de memoria.

El controlador de memoria es el dispositivo que realmente gobierna el sistema de memoria. Cuando el procesador ARM7TDMI quiere hacer un acceso a la memoria escribe en el bus la dirección a la que quiere acceder, el controlador de memoria se encargará de generar las señales adecuadas para acceder al chip de memoria externo que corresponda según la dirección puesta en el bus.

El controlador estructura el espacio de direcciones del ARM7TDMI como se ilustra en la figura 3.4. Como podemos ver el controlador de memoria reduce el espacio útil de 4GB a 256MBytes, que queda organizado en 8 fragmentos de 32MBytes denominados bancos. Cada Banco podrá ser asignado a un chip de memoria externo distinto, aunque sólo los dos últimos bancos admiten cualquier tipo de memoria. El controlador de memoria generará las señales de activación adecuadas (*chip select*, etc.) para distintos chips de memoria externos asociados a cada uno de los bancos.

De todo el espacio de memoria, la parte alta del primer banco, correspondiente al rango `0x01C00000-0x01FFFFFF`, está reservada para los registros internos del S3C44BOX. Cuando el procesador realiza un acceso en este rango el controlador de memoria se inhibe (no

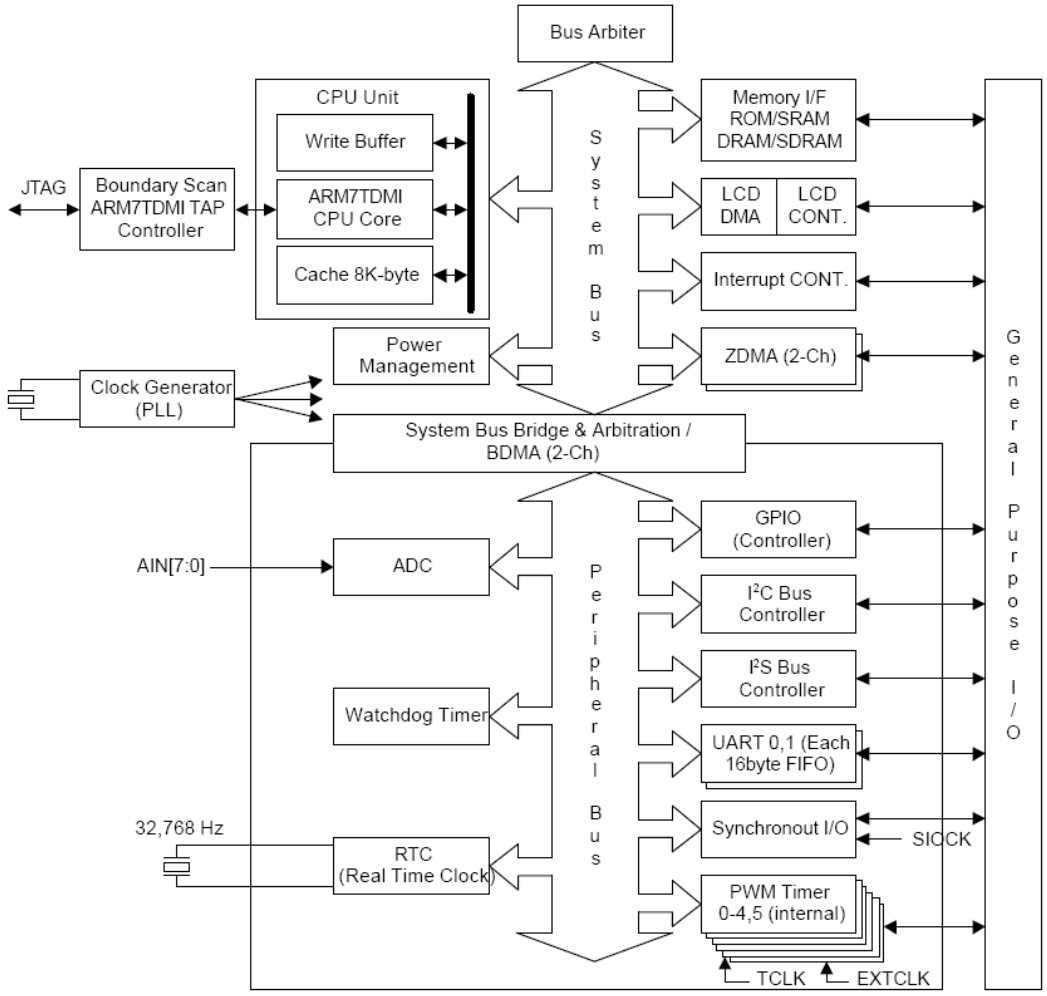

Figura 3.3: Estructura interna del *System on chip* S3C44BOX de Samsung.

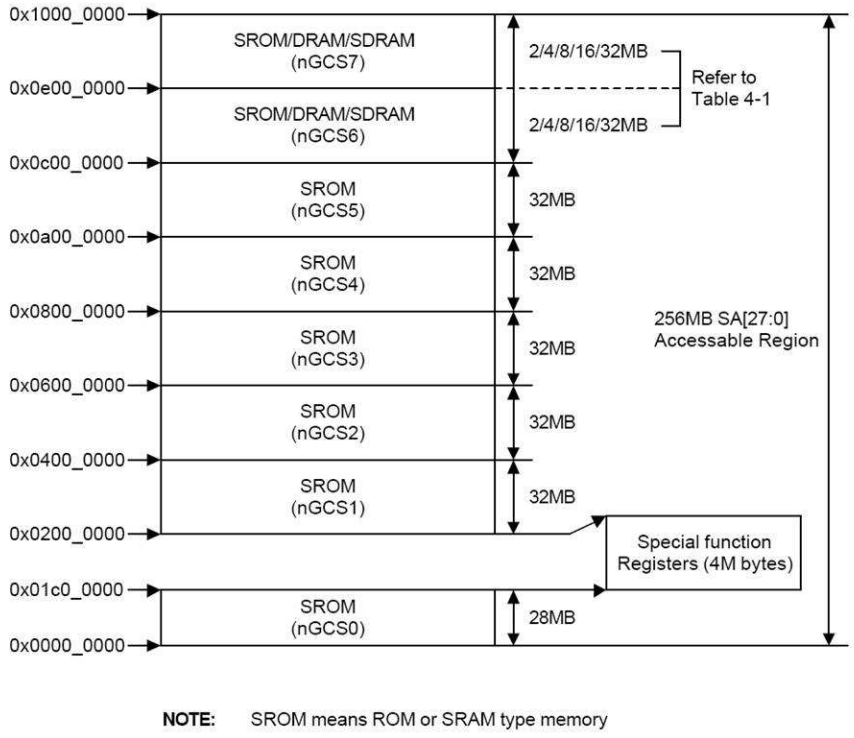

Figura 3.4: Mapa de memoria del S3C44B0X. Las señales nCGS* son generadas por el controlador de memoria cuando se accede a una dirección en el rango perteneciente a cada uno de los bancos. Estas señales se utilizan externamente para activar los distintos componentes ubicados en cada banco, por ejemplo las entradas *chip select* de los chips de memoria.

genera ninguna señal) y el controlador interno correspondiente será accedido. El controlador de memoria también tiene sus registros de control, trece en total, ubicados en este rango. Estos registros permiten configurar el controlador de acuerdo a los chips de memoria externos que tenga conectados en cada banco, y así poder generar sus señales de control:

BWSCON Permite configurar el ancho del bus y el estado de espera (*Wait Status*).

BANKCON0-7 Principalmente permiten configurar la temporización de cada banco (ciclos de setup, chip select, acceso, ...). Los valores que deben escribirse dependen obviamente del tipo de memoria conectada a cada banco.

REFRESH Configura el refresco de la memoria.

BANKSIZE Permite la configuración del tamaño del banco.

MRSRB6-7 Configura parámetros de distintos modos de acceso (latencia de CAS, modo ráfaga, modo test) – sólo para memorias DRAM y SDRAM –.

3.3.3. La S3CEV40

La placa que se emplea en el laboratorio (S3CEV40) dispone de dos memorias de distinto tipo:

- Memoria ROM Flash de 1Mx16bits, ubicada en el banco 0 del S3C44B0X y cuyo rango de direcciones es `[0x00000000-0x001FFFF]`. La figura 3.5 ilustra el esquema de conexión entre el chip S3C44B0X y la ROM.
- Memoria SDRAM de 4Mx16bits, ubicada en el banco 6 del S3C44B0X y cuyo rango de direcciones es `[0x0C000000-0x0C7FFFFF]`. La figura 3.6 ilustra el esquema de conexión entre el chip S3C44B0X y la SDRAM.

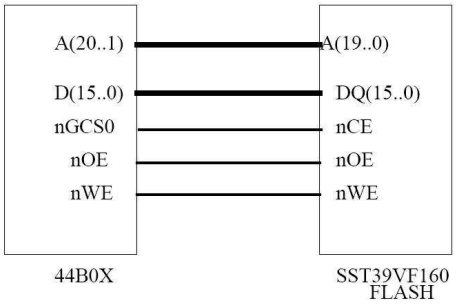

Figura 3.5: Conexión entre el S3C44B0X y la ROM Flash de la placa .

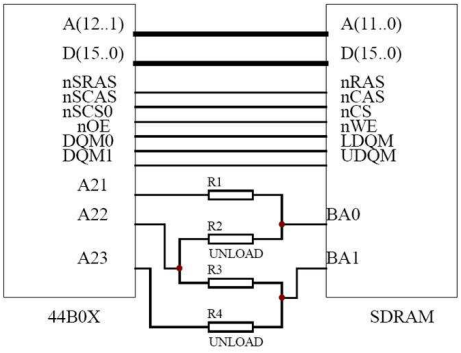

Figura 3.6: Conexión entre el S3C44B0X y la SDRAM de la placa

Como podemos ver, el espacio de 256MB total permitido por el controlador de memoria se reduce en nuestro caso a 10MB, 2 de ROM-FLASH y 8 de SDRAM.

Finalmente, debemos señalar que los dispositivos de E/S incluidos en la placa, se encuentran ubicados fuera del rango de direcciones asociado a estas memorias. Estudiaremos el sistema de E/S en detalle en las próximas prácticas.

3.4. Inicialización del controlador de memoria

Como hemos visto en la sección 3.3 el controlador de memoria es el encargado de, en función de la dirección puesta en el bus del sistema, generar las señales que permiten acceder

al chip de memoria correspondiente al banco accedido. Además los distintos bancos pueden tener conectados chips de memoria de distintas tecnologías, como por ejemplo en nuestro caso un chip FLASH y un chip SDRAM. Por ello, antes de utilizar el sistema de memoria debe configurarse el controlador de memoria de manera adecuada. La configuración del controlador de memoria se lleva a cabo a través de sus 13 registros de control, descritos en la sección 3.3.2.

En el caso del laboratorio, con la configuración descrita en la sección 3.3.3, los valores que debemos asignar a cada uno de los registros aparecen en la tabla 3.4. Para conocer exactamente el significado de estos valores debe consultarse el Manual de Referencia del S3C44BOX [um-].

Tabla 3.4: Configuración del controlador de memoria.

Registro	**Dirección**	**Valor**
BWSCON	0x01C80000	0x11110102
BANKCON0	0x01C80004	0x00000600
BANKCON1	0x01C80008	0x00007FFC
BANKCON2	0x01C8000C	0x00007FFC
BANKCON3	0x01C80010	0x00007FFC
BANKCON4	0x01C80014	0x00007FFC
BANKCON5	0x01C80018	0x00007FFC
BANKCON6	0x01C8001C	0x00018000
BANKCON7	0x01C80020	0x00018000
REFRESH	0x01C80024	0x00860459
BANKSIZE	0x01C80028	0x00000010
MRSRB6	0x01C8002C	0x00000020
MRSRB7	0x01C80030	0x00000020

3.5. Interacción con el Embest IDE mediante comandos

Además de mediante la interfaz gráfica, el entorno de desarrollo de Embest permite realizar algunas de las funciones de depuración mediante comandos de texto. Los comandos más usados son los siguientes:

1. `help`: Muestra la ayuda de cada comando.
2. `go`: Ejecuta el programa cargado en memoria a partir de la dirección actual del contador de programa.
3. `memwrite [opción] dirección valor`: Escribe el valor indicado en la dirección de memoria especificada. Los accesos a memoria son por defecto de tamaño palabra y usan ordenación *Little Endian*. Las opciones básicas son:

   -h Especifica el acceso a memoria en tamaño de media palabra.

   -b Especifica el acceso a memoria en tamaño byte.

   -e Escribe en memoria usando ordenación *Big Endian*.

Ejemplo: `memwrite 0x1000 0x5A ;escribir 0x0000005A en 0x1000`

4. `refresh`: Actualizar la información mostrada en las ventanas del entorno IDE. Las ventanas que se actualizan son la ventana de los registros, la ventana de la memoria, la ventana de la pila, la ventana de visualización de monitorización (*Watch*) y la ventana de variables/símbolos globales y locales.

5. `regwrite Rx valor`: Escribe el valor indicado en un registro de la arquitectura ARM (Rx).
   Ejemplo: `regwrite pc 0x3840 ;escribe el valor 0x3840 en el contador de programa`

6. `reset`: Reinicializa el procesador ARM7.

7. `script`: Ejecuta un fichero de comandos.

8. `stop`: Para la ejecución del procesador ARM7.

Estos comandos se pueden emplear de dos formas distintas, directamente en la ventana `Command` o formando parte de un fichero o *script* de depuración. En este último caso, cada comando se debe escribir en una línea distinta, donde la presencia del símbolo ";" indica el final del comando en sí y el comienzo de los comentarios. Los comandos que se encuentren contenidos dentro de un *script* de depuración se ejecutan automáticamente en estricto orden secuencial. Pondremos a este tipo de ficheros la extensión `.cs` y para ejecutarlos hay dos procedimientos básicos:

- Mediante el comando `script` en la ventana de comandos del entorno IDE.

- Mediante las opciones del proyecto. En la pestaña de depuración (*Debug*) se puede indicar un script que se ejecutará tras haberse conectado a la placa *"Action After Connected"*. Esta facilidad es tremendamente útil, por ejemplo para preparar el sistema, dejándolo en el estado que queremos antes de subir el programa.

En la primera parte de la práctica vamos a utilizar un script de este tipo para preparar el sistema para la ejecución del programa. En concreto, tendremos que hacer que el script pare el procesador, produzca un reset y configure correctamente el controlador de memoria, utilizando los valores de la tabla 3.4.

3.6. Gestión de excepciones en la placa S3CEV40

Como hemos mencionado arriba, las excepciones en el ARM7TDMI son autovectorizadas, es decir, el vector de interrupción se genera de forma automática en función de la excepción (ver tabla 3.2). Por lo tanto cuando se produce una excepción el procesador ejecuta la instrucción que está almacenada en memoria en la dirección indicada por el vector. Podemos comprobar en la tabla 3.2 que los vectores corresponden a direcciones del comienzo del mapa de memoria, situadas en un chip flash de la placa.

La memoria flash de la placa del laboratorio contiene un programa de test suministrado por el fabricante. Este programa comienza con un código similar al siguiente:

```
.equ  _ISR_STARTADDRESS,0xc7fff00 /* GCS6:64M DRAM/SDRAM  */

.equ    UserStack,   _ISR_STARTADDRESS-0xf00         /* c7ff000 */
.equ    SVCStack,    _ISR_STARTADDRESS-0xf00+256     /* c7ff100 */
.equ    UndefStack,  _ISR_STARTADDRESS-0xf00+256*2   /* c7ff200 */
.equ    AbortStack,  _ISR_STARTADDRESS-0xf00+256*3   /* c7ff300 */
.equ    IRQStack,    _ISR_STARTADDRESS-0xf00+256*4   /* c7ff400 */
.equ    FIQStack,    _ISR_STARTADDRESS-0xf00+256*5   /* c7ff500 */

.equ    HandleReset,    _ISR_STARTADDRESS
.equ    HandleUndef,    _ISR_STARTADDRESS+4
.equ    HandleSWI,      _ISR_STARTADDRESS+4*2
.equ    HandlePabort,   _ISR_STARTADDRESS+4*3
.equ    HandleDabort,   _ISR_STARTADDRESS+4*4
.equ    HandleReserved, _ISR_STARTADDRESS+4*5
.equ    HandleIRQ,      _ISR_STARTADDRESS+4*6
.equ    HandleFIQ,      _ISR_STARTADDRESS+4*7

.macro HANDLER HandleLabel
    sub   sp,sp,#4         /* Decrementamos sp en 4 */
    stmfd sp!,{r0}         /* Salvamos r0 en la pila */
    ldr   r0,=\HandleLabel /* Cargamos en r0 el valor del símbolo pasado como
                              argumento a la macro */
    ldr   r0,[r0]          /* Cargamos en r0 el contenido de esta dirección,
                              que será la dirección de comienzo de la rutina
                              de tratamiento de la excepción */
    str   r0,[sp,#4]       /* Almacenamos esta dirección en la posición de la
                              pila que reservamos al comienzo */
    ldmfd sp!,{r0,pc}      /* Saltamos a la dirección restaurando r0 */
.endm

/*Comienzo del programa */
start:
    b ResetHandler          /* 0x00 */
    b HandlerUndef          /* 0x04 */
    b HandlerSWI            /* 0x08 */
    b HandlerPabort         /* 0x0C */
    b HandlerDabort         /* 0x10 */
    b .                     /* 0x14 */
    b HandlerIRQ            /* 0x18 */
    b HandlerFIQ            /* 0x1C */

    /*Más código que veremos en la práctica siguiente */
    .align
```

```
    HandlerFIQ:     HANDLER HandleFIQ
    HandlerIRQ:     HANDLER HandleIRQ
    HandlerUndef:   HANDLER HandleUndef
    HandlerSWI:     HANDLER HandleSWI
    HandlerDabort:  HANDLER HandleDabort
    HandlerPabort:  HANDLER HandlePabort

    /*Más código que veremos en la práctica siguiente */

ResetHandler:
    /* Código de la rutina de reset */
```

Como vemos para la excepción *Reset* la instrucción en la dirección indicada por el vector (`0x00`) realiza un salto relativo al PC a la dirección de la rutina de tratamiento de excepciones que está almacenada en la flash. Por lo tanto, a menos que modifiquemos el contenido de la flash no podremos sustituir esta rutina.

El resto de excepciones son tratadas de otro modo. La instrucción en la dirección indicada por el vector salta a una posición en la flash en la que hay un pequeño fragmento de código generado por la macro `HANDLER`. Si analizamos el código de esta macro veremos que, después de salvar en la pila el registro r0 para conservar su valor, lee de memoria el valor del símbolo que se pasó como argumento a la macro. El valor del símbolo es la dirección de memoria donde espera encontrar la dirección de comienzo de la rutina de tratamiento de la excepción.

Por lo tanto se lee de memoria la dirección de la rutina de tratamiento de interrupciones, almacenada en una posición definida, y realiza un salto a esta dirección. Podemos ver además que las posiciones de memoria donde se almacenan las direcciones de las rutinas (HandleFIQ, HandleIRQ, ...) comienzan en la dirección dada por el símbolo `_ISR_STARTADDRESS`: `0xc7fff00`. El proceso se ilustra en la figura 3.7.

Por ejemplo, supongamos que se produce una excepción Undef. En este caso se ejecuta la instrucción en la dirección `0x04`. Esta instrucción realiza un salto a la dirección de la memoria flash etiquetada con `HandlerUndef`. En esta dirección estaría el siguiente código:

```
    sub     sp,sp,#4
    stmfd   sp!,{r0}
    ldr     r0,=(_ISR_STARTADDRESS+4) /* 0x0c7fff04 */
    ldr     r0,[r0]
    str     r0,[sp,#4]
    ldmfd   sp!,{r0,pc}
```

que lee de memoria lo que hay en la posición `0x0c7fff04`, lo toma como dirección destino del salto (realizado mediante una instrucción `ldm`).

Concluyendo, si queremos tratar las excepciones (salvo Reset) sin tener que modificar el contenido de la flash, debemos escribir en la tabla que comienza en la dirección `0x0c7fff00` las direcciones de comienzo de nuestras rutinas de tratamiento de excepción.

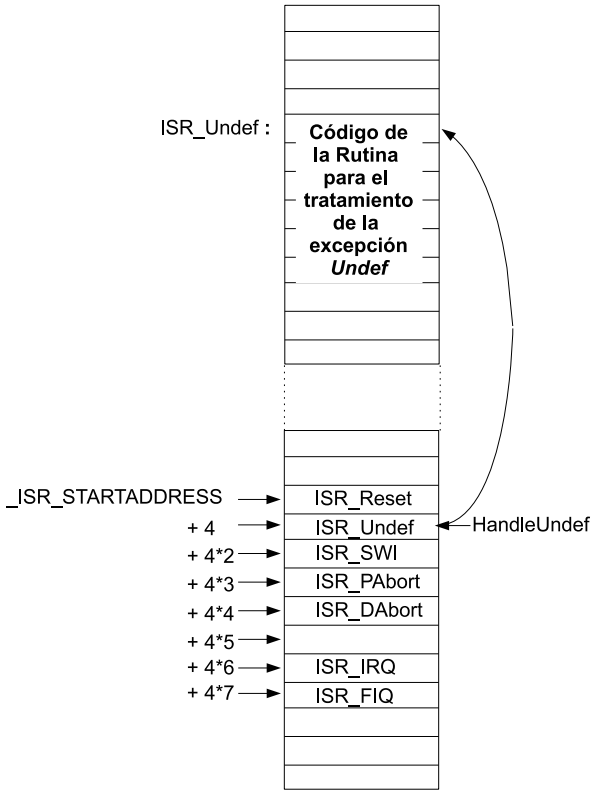

Figura 3.7: Tabla de direcciones de las rutinas para el tratamiento de excepciones.

3.7. Mapa de memoria de un programa en las prácticas

De aquí en adelante vamos a utilizar un mapa de memoria muy similar para todos los programas. Este mapa está ilustrado en la figura 3.8. Como vemos las direcciones altas del banco se reservan para almacenar la tabla de direcciones de las rutinas de tratamiento de excepciones. Justo por encima (direcciones anteriores) se reserva un espacio para las pilas de los distintos modos de ejecución (generalmente inicializadas por el programa residente en la flash). El espacio restante será utilizado para ubicar las distintas secciones de nuestro programa (`text`, `data`, `rodata` y `bss`) comenzando en la dirección más baja del banco 6. El espacio restante será utilizado como *heap*, para asignarlo dinámicamente a través de llamadas a *malloc*. La gestión del *heap* es responsabilidad del propio programa. En las prácticas siguientes suministraremos una biblioteca que permitirá hacer un uso muy rudimentario de este *heap*.

3.8. Desarrollo de la Práctica

En esta práctica vamos a dividir el trabajo en dos partes, ambas parcialmente guiadas. En la primera pondremos en práctica los conocimientos expuestos en los apartados anteriores sin modificar la memoria flash. En la segunda realizaremos modificaciones sobre el código de la primera para conseguir un programa residente en flash capaz de ejecutarse desde la SDRAM.

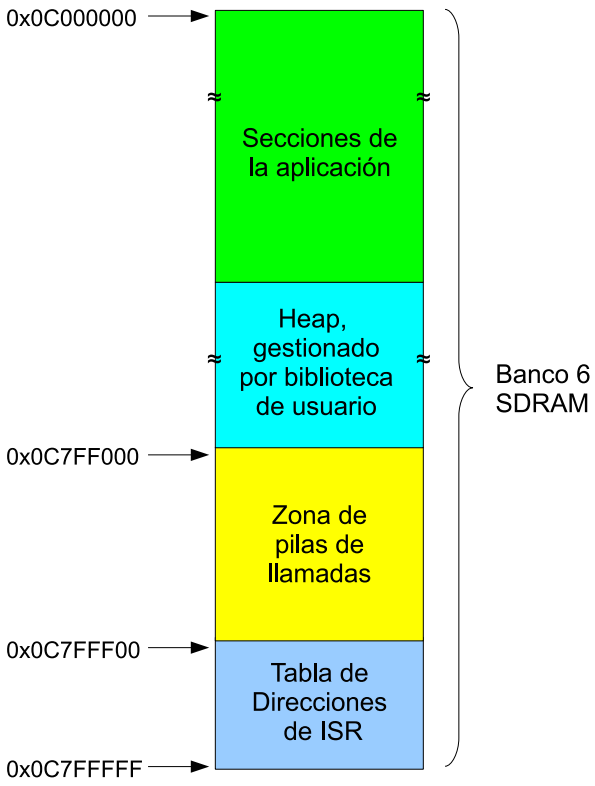

Figura 3.8: Mapa de memoria utilizado para los programas de las prácticas.

3.8.1. Primera parte

Vamos a crear un pequeño proyecto que inicialice la placa mediante un script, cargue un programa en memoria en la dirección de comienzo de la SDRAM y lo ejecute. El programa contendrá un código en ensamblador que inicialice correctamente el estado del proceso para todos los modos de ejecución y configure la tabla de direcciones de las rutinas de tratamiento de excepciones. Al finalizar ejecutará la función Main, que generará tres excepciones (Undef, Dabort y SWI) y terminará. Para ello seguimos los siguientes pasos:

1. Creamos un workspace nuevo y le añadimos una carpeta `common`.

2. Creamos un fichero nuevo y lo guardamos con el nombre `ev40boot.cs`. Lo añadimos a la carpeta `common`. Este fichero lo vamos a utilizar como script para la configuración del sistema. Debemos configurar el proyecto para que ejecute el script al conectarse a la placa (casilla *"Action After Connected"* de la pestaña Debug/General). Siguiendo las explicaciones de la sección 3.5 deberíamos crear el siguiente script:

```
stop                          ; stop the processor
reset                         ; reset board
memwrite 0x01D30000 0x00000000 ; WTCON  disable watchdog
memwrite 0x01E0000C 0x07ffffff ; INTMSK (disable all interrupt)
memwrite 0x01E00024 0xffffffff ; clear all interrupt
memwrite 0x01C80000 0x11110102 ; BWSCON
memwrite 0x01C80004 0x00000600 ; BANKCON0
memwrite 0x01C80008 0x00007FFC ; BANKCON1
memwrite 0x01C8000C 0x00007FFC ; BANKCON2
memwrite 0x01C80010 0x00007FFC ; BANKCON3
memwrite 0x01C80014 0x00007FFC ; BANKCON4
memwrite 0x01C80018 0x00007FFC ; BANKCON5
memwrite 0x01C8001C 0x00018000 ; BANKCON6
memwrite 0x01C80020 0x00018000 ; BANKCON7
memwrite 0x01C80024 0x00860459 ; REFRESH
memwrite 0x01C80028 0x00000010 ; BANKSIZE
memwrite 0x01C8002C 0x00000020 ; MRSRB6
memwrite 0x01C80030 0x00000020 ; MRSRB7
```

3. Creamos un fichero nuevo y lo guardamos con el nombre `ldscript.ld`. Lo añadimos a la carpeta `common`. Como de costumbre será el script de configuración del enlazador. Tiene que colocar la sección de código a partir de la posición donde comienza la SDRAM (`0x0C000000`). Las secciones de datos las pondremos justo después. Como en la práctica anterior, aprovecharemos para definir una serie de símbolos que nos permitan saber dónde empieza y dónde termina cada sección. Por lo tanto el contenido del fichero será:

```
SECTIONS
{
. = 0x0C000000;
Image_RO_Base = .;
```

```
.text : { *(.text) }
Image_RO_Limit = .;
Image_RW_Base = .;
.data : { *(.data) }
.rodata : { *(.rodata) }
Image_RW_Limit = .;
Image_ZI_Base = .;
.bss : { *(.bss) }
Image_ZI_Limit = .;
__bss_start__ = .;
__bss_end__ = .;
```

4. Crearemos un fichero nuevo que guardaremos con el nombre `init.s`. Este fichero contendrá el código que debe inicializar la arquitectura para luego poder ejecutar nuestro programa. Como la inicialización del controlador de memoria se realiza mediante el script `ev40boot.cs`, el código de `init.s` no se tendrá que preocupar de ello. Se encargará de inicializar el registro de pila para cada uno de los modos de ejecución del procesador y escribir en la tabla de direcciones de rutinas de tratamiento de excepción la dirección de nuestras rutinas. Después pasará la ejecución a modo usuario, inicializará su pila y el frame pointer y ejecutará la función `Main`. El código parcialmente completo es el siguiente:

```
.global DoUndef
.global DoSWI
.global DoDabort
.global screen

.global _ISR_STARTADDRESS
.equ  _ISR_STARTADDRESS,0xc7fff00 /* GCS6:64M DRAM/SDRAM  */
.equ    UserStack,   _ISR_STARTADDRESS-0xf00         /* c7ff000 */
.equ    SVCStack,    _ISR_STARTADDRESS-0xf00+256     /* c7ff100 */
.equ    UndefStack,  _ISR_STARTADDRESS-0xf00+256*2   /* c7ff200 */
.equ    AbortStack,  _ISR_STARTADDRESS-0xf00+256*3   /* c7ff300 */
.equ    IRQStack,    _ISR_STARTADDRESS-0xf00+256*4   /* c7ff400 */
.equ    FIQStack,    _ISR_STARTADDRESS-0xf00+256*5   /* c7ff500 */

.equ    HandleReset,    _ISR_STARTADDRESS
.equ    HandleUndef,    _ISR_STARTADDRESS+4
.equ    HandleSWI,      _ISR_STARTADDRESS+4*2
.equ    HandlePabort,   _ISR_STARTADDRESS+4*3
.equ    HandleDabort,   _ISR_STARTADDRESS+4*4
.equ    HandleReserved, _ISR_STARTADDRESS+4*5
.equ    HandleIRQ,      _ISR_STARTADDRESS+4*6
.equ    HandleFIQ,      _ISR_STARTADDRESS+4*7

/* Símbolos para facilitar la codificación
de los modos de ejecución */
```

```
.equ  USERMODE,0x10
.equ  FIQMODE,0x11
.equ  IRQMODE,0x12
.equ  SVCMODE,0x13
.equ  ABORTMODE,0x17
.equ  UNDEFMODE,0x1b
.equ  MODEMASK,0x1f

.equ  NOINT,0xc0   /* Máscara para deshabilitar interrupciones */
.equ    IRQ_MODE,0x40   /* deshabilitar interrupciones en modo IRQ  */
.equ    FIQ_MODE,0x80   /* deshabilitar interrupciones en modo FIQ  */

.arm
start:
    /* Si comenzamos con un reset
     el procesador está en modo supervisor */
    bl InitStacks

    /* Seguimos en modo supervisor, configuramos
       las direcciones de las rutinas de tratamiento
       de excepciones */
    bl InitISR

    /* Pasamos a modo usuario, inicializamos su pila
      y ponemos a cero el frame pointer*/
    mrs     r0,cpsr
    bic r0,r0,#MODEMASK
    orr r1,r0,#USERMODE
    msr cpsr_cxsf,r1      /* UserMode */
    ldr sp,=UserStack
    mov fp,#0

    /* Saltamos a Main */
    .extern Main

    ldr r0,=Main
    mov lr,pc
    mov pc,r0
    b .
InitStacks:
    /* Añadir código para configurar
    los registros de pila de cada modo.
    Para ello hay que pasar al modo y escribir
    en sp la dirección de comienzo de la pila
    para ese modo. Estas direcciones están dadas
    por los símbolos definidos arriba */
```

```
    mov pc, lr

InitISR:
    /* Añadir código para configurar
    las direcciones de comienzo de las rutinas
    de tratamiento de excepción. Por ejemplo
    para Undef hay que copiar la dirección de
    ISR_Undef en la dirección HandleUndef */

mov pc,lr

/* Rutinas de generación de excepciones */
DoSWI:
    swi
    mov pc,lr

DoUndef:
    .word 0xE6000010
    mov pc,lr

DoDabort:
    ldr r0,=0x0a333333
    str r0,[r0]
    mov pc,lr


screen:
    .space 1024

.end
```

5. Creamos un fichero `main.c` en el que añadimos la función Main y las rutinas de tratamiento de excepción programadas en C. El código completo viene a continuación:

```
extern char screen[];

char *Screen = (char*) screen;

/* Declaración de funciones para que el compilador las
 genere como rutinas de tratamiento de excepción */
void ISR_SWI(void) __attribute__ ((interrupt ("SWI")));
void ISR_Undef(void) __attribute__ ((interrupt ("UNDEF")));
void ISR_IRQ(void) __attribute__ ((interrupt ("IRQ")));
void ISR_FIQ(void) __attribute__ ((interrupt ("FIQ")));
void ISR_Pabort(void) __attribute__ ((interrupt ("ABORT")));
void ISR_Dabort(void) __attribute__ ((interrupt ("ABORT")));
```

```
void write(char* text, char* address)
{
while( *text != 0 ){
*address++ = *text++;
}
}

void DoUndef();
void DoDabort();
void DoSWI();

void Main(void)
{
    DoUndef();
    DoSWI();
    DoDabort();
}

void ISR_Undef(void)
{
    write("Undef ",Screen);
}

void ISR_IRQ(void)
{
    write("IRQ   ",Screen);
}

void ISR_FIQ(void)
{
    write("FIQ   ",Screen);
}

void ISR_SWI(void)
{
    write("SWI   ",Screen);
}

void ISR_Pabort(void)
{
    write("Pabort",Screen);
}

void ISR_Dabort(void)
{
    write("Dabort",Screen);
}
```

6. Compilar el proyecto y subirlo a la placa. Ejecutar paso a paso el programa para comprobar que funciona correctamente. De lo contrario debemos corregirlo.

7. Contestad a las siguientes preguntas:

   - ¿Cómo se generan las excepciones Undef, Dabort y SWI?
   - ¿Qué hacen las rutinas de tratamiento de estas excepciones?
   - Observar el código generado para estas rutinas (desensamblado). ¿En qué se diferencia de una rutina corriente? Detallar la respuesta.

8. Presionar el reset de la placa. ¿Qué sucede? ¿Qué código se ejecuta?

9. **Ejercicio:** Modificar el código codificando la rutina write en ensamblador.

3.8.2. Segunda parte

Vamos a modificar el programa anterior para que resida en la flash, sustituyendo al programa de test actual. La idea es que al resetear el sistema se ejecute la rutina de tratamiento de Reset, que inicializará el sistema (lo que hacía el script `ev40boot.cs`) y una vez inicializado se copiará a sí mismo de la flash a la SDRAM para continuar allí su ejecución. Ya en la SDRAM hará lo que hacía en la primera parte. Claro está que para que el código de la primera parte siga funcionando debemos generar correctamente el código de tratamiento de las excepciones, de forma que siga leyendo la dirección de la rutina correspondiente en los mismos sitios que antes.

Procedemos de la siguiente manera:

1. Crear un nuevo proyecto, con los mismos ficheros que en la primera parte salvo el script `ev40boot.cs`. Configurarlo como en la primera parte excepto en el apartado `Debug`, donde marcamos la casilla de ninguna acción tras la conexión. En la zona de descarga podemos dejar todos los campos vacíos puesto que no lo vamos a descargar con el IDE sino con el programador de la memoria flash.

2. Modificar el fichero `init.s` copiando la estructura descrita en la sección 3.6 y añadiendo código a la rutina `ResetHandler`. Este manejador se encargará de inicializar la placa, copiar el programa a la SDRAM y pasar la ejecución a la SDRAM, donde inicializará las pilas de los distintos modos y saltará a ejecutar la función Main. La estructura del código será:

```
.global _ISR_STARTADDRESS
.global DoUndef
.global DoSWI
.global DoDabort
.global screen

.equ  _ISR_STARTADDRESS,0xc7fff00 /* GCS6:64M DRAM/SDRAM  */
.equ    UserStack,    _ISR_STARTADDRESS-0xf00         /* c7ff000 */
.equ    SVCStack,     _ISR_STARTADDRESS-0xf00+256     /* c7ff100 */
.equ    UndefStack,   _ISR_STARTADDRESS-0xf00+256*2   /* c7ff200 */
```

```
.equ    AbortStack,  _ISR_STARTADDRESS-0xf00+256*3   /* c7ff300 */
.equ    IRQStack,    _ISR_STARTADDRESS-0xf00+256*4   /* c7ff400 */
.equ    FIQStack,    _ISR_STARTADDRESS-0xf00+256*5   /* c7ff500 */

.equ    HandleReset,    _ISR_STARTADDRESS
.equ    HandleUndef,    _ISR_STARTADDRESS+4
.equ    HandleSWI,      _ISR_STARTADDRESS+4*2
.equ    HandlePabort,   _ISR_STARTADDRESS+4*3
.equ    HandleDabort,   _ISR_STARTADDRESS+4*4
.equ    HandleReserved, _ISR_STARTADDRESS+4*5
.equ    HandleIRQ,      _ISR_STARTADDRESS+4*6
.equ    HandleFIQ,      _ISR_STARTADDRESS+4*7

/* Simbolos para facilitar la codificación
de los modos de ejecución */
.equ  USERMODE,0x10
.equ  FIQMODE,0x11
.equ  IRQMODE,0x12
.equ  SVCMODE,0x13
.equ  ABORTMODE,0x17
.equ  UNDEFMODE,0x1b
.equ  MODEMASK,0x1f

.equ    NOINT,    0xc0   /* Máscara para deshabilitar interrupciones */
.equ    IRQ_MODE, 0x40   /* deshabilitar interrupciones en modo IRQ  */
.equ    FIQ_MODE, 0x80   /* deshabilitar interrupciones en modo FIQ  */

/* Direcciones de registros para inicialización */
.equ  INTMSK,0x01e0000c
.equ  WTCON,0x01d30000

/* símbolos del ldconfig.ld */
.extern Image_RO_Limit /* Final del código ROM (comiezo de los datos en ROM)*/
.extern Image_RW_Base  /* Base of RAM to initialise */
.extern Image_ZI_Base  /* Base and limit of area */
.extern Image_ZI_Limit /* to zero initialise */

.macro HANDLER HandleLabel
    sub   sp,sp,#4         /* Decrementamos sp en 4 */
    stmfd sp!,{r0}         /* Salvamos r0 en la pila */
    ldr   r0,=\HandleLabel /* Cargamos en r0 el valor del símbolo pasado como
                              argumento a la macro */
    ldr   r0,[r0]          /* Cargamos en r0 el contenido de esta dirección,
                              que será la dirección de comienzo de la rutina
                              de tratamiento de la excepción */
    str   r0,[sp,#4]       /* Almacenamos esta dirección en la posición de la
                              pila que reservamos al comienzo */
```

```
    ldmfd sp!,{r0,pc}      /* Saltamos a la dirección restaurando r0 */
.endm

.arm
start:
b ResetHandler          /* 0x00 */
b HandlerUndef          /* 0x04 */
b HandlerSWI            /* 0x08 */
b HandlerPabort         /* 0x0C */
b HandlerDabort         /* 0x10 */
b .                     /* 0x14 */
b HandlerIRQ            /* 0x18 */
b HandlerFIQ            /* 0x1C */

.align

HandlerFIQ: HANDLER HandleFIQ
HandlerIRQ: HANDLER HandleIRQ
HandlerUndef: HANDLER HandleUndef
HandlerSWI: HANDLER HandleSWI
HandlerDabort: HANDLER HandleDabort
HandlerPabort: HANDLER HandlePabort


ResetHandler:
    /* Deshabilitar el watchdog */

    /* Deshabilitar las interrupciones y borrar todas las pendientes*/

    /* Establecer los valores de los registros del controlador de memoria.
       Observar que los registros tienen direcciones consecutivas desde la
       dirección 0x01c80000 y que los valores a copiar los hemos almacenado
       en memoria a partir de la dirección dada por la etiqueta SDRAM.
       OJO: SDRAM es resuelto por el enlazador como si el programa comenzase
       en la dirección 0x0C000000, pero lo hemos cargado en la flash a partir
       de la dirección 0x00000000.
     */

    /* Copiar todo lo que hay desde el comienzo de la flash
       a la memoria SDRAM. Ayuda: tomar un puntero a Image_RO_Base e ir
       copiando lo que hay hasta que la posición destino coincida con
       Image_ZI_Limit */


ldr pc,=Init

Init:
```

```
    nop
    nop
    nop

    /* Si comenzamos con un reset
       el procesador esta en modo supervisor */
    bl InitStacks

    /* Seguimos en modo supervisor, configuramos
       las direcciones de las rutinas de tratamiento
       de excepciones */
    bl InitISR

    /* Pasamos a modo usuario, inicializamos su pila
       y ponemos a cero el frame pointer*/
    mrs     r0,cpsr
    bic r0,r0,#MODEMASK
    orr r1,r0,#USERMODE
    msr cpsr_cxsf,r1      /* SVCMode */
    ldr sp,=UserStack
    mov fp,#0

    /* Saltamos a Main */
    .extern Main

    ldr r0,=Main
    mov lr,pc
    mov pc,r0
    b .
InitStacks:
    /* Código de la primera parte */

InitISR:
    /* Código de la primera parte */

DoSWI:
    swi
    mov pc,lr

DoUndef:
    .word 0xE6000010
    mov pc,lr

DoDabort:
    ldr r0,=0x0a333333
    str r0,[r0]
    mov pc,lr
```

```
SMRDATA:
    .word 0x11110102 @ BWSCON
    .word 0x00000600 @ BANKCON0
    .word 0x00007FFC @ BANKCON1
    .word 0x00007FFC @ BANKCON2
    .word 0x00007FFC @ BANKCON3
    .word 0x00007FFC @ BANKCON4
    .word 0x00007FFC @ BANKCON5
    .word 0x00018000 @ BANKCON6
    .word 0x00018000 @ BANKCON7
    .word 0x00860459 @ REFRESH
    .word 0x00000010 @ BANKSIZE
    .word 0x00000020 @ MRSRB6
    .word 0x00000020 @ MRSRB7

screen:
    .space 1024

    .end
```

3. Construir el proyecto.

4. En el menú `Tools` seleccionamos `Elf to Bin` para crear un fichero binario a partir del fichero ELF creado por el enlazador.

5. Creamos un fichero con el nombre `EmbestS3CEV40.cfg`. Este fichero contendrá la configuración de la placa para el programador de la memoria flash. Copiamos el siguiente contenido:

```
[SETUP]
CpuVendor=Samsung
CpuChip=S3C44B0X
FlashVendor=SiliconStorageTechnology
FlashChip=SST39LF/VF160
RamAddress=$0C000000
RamSupport=1
FlashAddress=$00000000
FlashWidth=16
FlashChipsPerSector=1
LittleEndian=0
SectStart=0
SectEnd=89
AutoErase=1
AutoVerify=1
CpuEndian=LITTLE
SimCount=14
MemoryCount=0
```

```
ProgramFile=
UploadFile=
Format=Flat Bin
Sim14=MRSRB7:$00000020
Sim13=MRSRB6:$00000020
Sim12=BANKSIZE:$00000010
Sim11=REFRESH:$00860459
Sim10=BANKCON7:$00018000
Sim9=BANKCON6:$00018000
Sim8=BANKCON5:$00007FFC
Sim7=BANKCON4:$00007FFC
Sim6=BANKCON3:$00007FFC
Sim5=BANKCON2:$00007FFC
Sim4=BANKCON1:$00007FFC
Sim3=BANKCON0:$00000600
Sim2=BWSCON:$11110101
Sim1=WTCON:$00000000
```

6. En el menú `Tools` seleccionamos `Flash Programmer`. Una vez abierto seleccionamos el menú `File` y abrimos el fichero `EmbestS3CEV40.cfg` creado anteriormente.

7. Escribimos un nombre de fichero en la casilla `Upload:`, por ejemplo `flash_upload.bin`. En este fichero copiaremos el contenido actual de la flash. Para ello hacemos click en el botón `Upload`.

8. Cuando termine escribimos el nombre del fichero `.bin` creado a partir del proyecto en la casilla `Program:`. Después hacemos click en el botón `Program`. Observemos que aunque el ELF lo hemos creado colocando el código a partir de la dirección `0x0C000000` (con el fichero de configuración del enlazador), al bajarlo a la placa lo estamos copiando a partir de la dirección `0x00000000`. Como consecuencia aunque el código se posicione en la memoria flash a partir de la dirección `0x00000000` todos los símbolos y etiquetas resueltos por el enlazador toman el valor como si el código comenzase en la dirección `0x0C000000`.

9. Una vez terminado el proceso de carga nos conectamos al depurador con el Embest IDE y paramos el procesador si es necesario.

10. En la consola de comandos ejecutamos la orden `reset` o pulsamos el botón rojo de reset del Embest IDE o el botón de reset de placa.

11. Ejecutamos paso a paso. Debemos ver que la ejecución comienza en la dirección cero (no habrá información de depuración como etiquetas). Podemos detectar el punto en que la ejecución debe pasar a la memoria SDRAM buscando las tres `nop` consecutivas. Cuando saltemos a la SDRAM aparecerán los símbolos de depuración. Debemos comprobar que la función Main se ejecuta con el mismo comportamiento que en la primera parte de la práctica.

12. **IMPORTANTE:** Antes de irnos del laboratorio debemos restaurar la imágen de la memoria flash. Para ello abrimos el programador de la flash, escribimos el

nombre que le dimos al fichero de upload en el paso 7 en la casilla `Program:`. Después hacemos click en el botón `Program` y esperamos a que se cargue correctamente.

Bibliografía

[arm] Arm architecture reference manual. Accesible en http://www.arm.com/miscPDFs/14128.pdf. Hay una copia en el campus virtual.

[um-] S3c44b0x risc microprocessor product overview. Accesible en http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=229&partnum=S3C44B0. Hay una copia en el campus virtual.