Práctica 2
Estructura de Computadores

Modos de ejecución, gestión de excepciones y entrada/salida mediante interrupciones

Índice general

2.1. Objetivos de la práctica

En esta práctica finalizaremos el estudio del procesador ARM7TDMI analizando sus modos de ejecución, sus excepciones y su sistema de entrada/salida, que gestionaremos mediante interrupciones. En la práctica comenzaremos a analizar algunas características del sistema en chip Samsung S3C44BOX y la placa Embest 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 entrada/salida.
- Conocer el mecanismo de entrada/salida mediante interrupciones.

Concretamente, el alumno deberá preparar un programa que inicialice correctamente las pilas de los distintos modos de ejecución del ARM y configure la tabla con las direcciones de las rutinas de tratamiento de excepción, pudiendo comprobar que se ejecutan correctamente

cuando se produce la excepción. En la segunda parte deberá modificar la práctica de entrada/salida controlada por programa, realizada en la asignatura de Fundamentos de Computadores de modo que funcione mediante interrupciones.

2.2. Excepciones y modos de ejecución

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 (habitualmente denominada *Interrupt Service Routine* (abreviado como *ISR*) o rutina de tratamiento de interrupciones) que tratará esa excepción. Esta rutina no puede ser diseñada como una subrutina corriente, siguiendo el AAPCS. Como la excepción es un evento no controlado por el programador, la rutina de tratamiento de la excepción debe preservar el estado del procesador completo, es decir los registros R0-R15[1] y el CPSR. Así, cuando finaliza el tratamiento de la excepción puede restaurarse el estado del procesador y retomar la ejecución del programa en el punto en que se dejó. Además, debemos tener en cuenta que pueden producirse varias excepciones simultáneamente, por lo que deberán establecerse prioridades a la hora de atenderlas.

Las excepciones en los procesadores de ARM son autovectorizadas. Esto quiere decir que cuando se produce una excepción, el procesador ejecuta automáticamente la instrucción ubicada en una dirección de memoria específica, que únicamente depende del tipo de excepción. A esta dirección se la denomina vector de la excepción (o interrupción). Normalmente esta instrucción no es más que un salto al comienzo de la rutina de tratamiento de la excepción.

El procesador ARM7TDMI tiene varios modos de ejecución, en su mayoría dedicados a atender excepciones. En la siguiente sección describimos estos modos.

2.2.1. Modos de Ejecución

Todos los programas desarrollados en las prácticas anteriores se ejecutaban en un modo de ejecución, aunque no nos hemos preocupado de ello. Sin embargo el ARM7TDMI dispone de 7 modos de ejecución diferentes, que permiten, entre otras cosas, la gestión eficiente de excepciones.

La figura 2.1 muestra los distintos campos del registro de estado. Los cinco bits menos significativos (`M[4:0]`) codifican el modo actual del procesador, por lo que cambiando estos bits el procesador cambia de modo. Sin embargo, la manera más habitual de cambiar de modo es a través de una excepción.

La tabla 2.1 describe los siete modos de ejecución del ARM7TDMI. Todos los modos excepto *User* (*usr*) son privilegiados. En los modos privilegiados no tenemos limitaciones de acceso a los recursos del procesador. En cambio, en los modos no privilegiados algunos recursos pueden estar restringidos. Concretamente, en el caso del ARM7TDMI los modos privilegiados

[1] El registro R15 no se preserva exactamente, en realidad se guarda la dirección en la que se interrumpió el programa en un registro especial para poder posteriormente reanudar su ejecución a partir de ese punto.

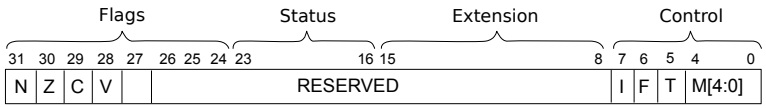

Figura 2.1: Descripción del Registro de estado (CPSR).

son los únicos en los que tenemos acceso no restringido al registro de estado. En modo *usr*, en cambio, no podemos modificar directamente (con la instrucción `mrs`) los bits de modo, y por tanto la única forma de cambiar de modo *usr* a cualquier otro modo es mediante una excepción.

Tabla 2.1: Modos del procesador

Modo del procesador	**Codigo**	**Uso**
usr	10000	Ejecución de código de usuario
fiq	10001	Servicio de int. rápidas
irq	10010	Servicio de int. lentas
svc	10011	Modo protegido para sistema operativo (int. sw)
abt	10111	Procesado de fallos de acceso a mem
und	11011	Manejo de instrucc. indefinidas
sys	11111	Ejecución de tareas del SO

De los modos privilegiados, cinco son conocidos como modos de excepción, debido a que están directamente relacionados con excepciones: FIQ (*fiq*), IRQ (*irq*), Supervisor (*svc*), Abort (*abt*) y Undef (*und*). El séptimo modo, System (*sys*) es diferente del resto de modos privilegiados, ya que el paso a este modo no ocurre mediante una excepción. Dicho modo 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

En las prácticas anteriores, trabajando en modo usuario, hemos manejado 15 registros de propósito general, el PC y el registro de estado CPSR. Sin embargo, la arquitectura dispone en realidad de 37 registros de 32 bits, incluyendo el contador de programa. Estos registros se organizan en bancos parcialmente solapados, y cada modo tiene asignado uno de los bancos, como ilustra la figura 2.2.

Debemos darnos cuenta de que en todos los bancos los registros de la parte superior solapan con los del primer banco, y por tanto son los mismos que los registros del modo usuario. Sin embargo, los modos FIQ, IRQ, Supervisor, Abort y Undefined tienen algunos registros propios, no solapados con los del modo usuario. Por ejemplo, cada uno tiene su propio puntero de pila SP (R13), lo que permite que cada modo utilice distintas zonas de memoria para la pila. Debemos inicializar el puntero de pila de cada uno de los modos.

Además, cada modo tiene su propio registro LR (R14). Veremos más adelante que cuando se produce una excepción el procesador cambia de modo y guarda en el LR del modo correspondiente la dirección de retorno, que servirá para retomar la ejecución del programa después de tratar la excepción. Además, cada modo, excepto el de usuario, dispone de su propio registro de sombra SPSR. Este registro se utiliza para salvar el registro de estado del programa

automaticamente cuando se produce una excepción y se cambia de modo de ejecución. Además, en el modo FIQ los registros R8-R12 son distintos de los del modo usuario, lo que facilita la preservación del contexto del programa (que corre en modo usuario). Finalmente, System es un modo privilegiado que utiliza los mismos registros que el modo usuario.

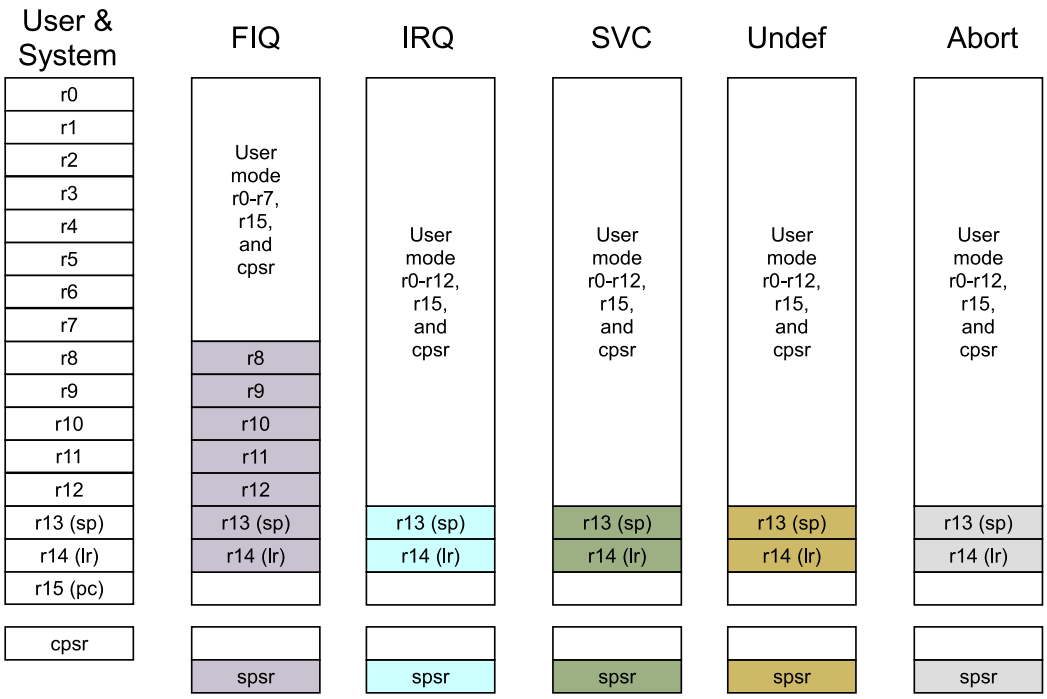

Figura 2.2: Registros visibles en cada modo de ejecución.

Como resumen, observemos que en cada modo podemos acceder a 15 registros de propósito general, llamados siempre r0-r14, el registro de estado CPSR, el registro de sombra SPSR (salvo en modo usuario) y el contador de programa PC (r15). Si un programa va a trabajar en varios modos de ejecución es necesario que inicialice el puntero de pila de cada uno de los modos que use. Por lo tanto, en las prácticas que utilicen excepciones habrá que inicializar no sólo el puntero de pila de usuario sino los de los otros modos de ejecución. Para acceder a un registro de un modo es necesario estar en ese 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. Éste 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.

El segundo mecanismo sólo está disponible cuando se está en un modo privilegiado. En este caso el cambio de modo puede realizarse escribiendo un valor adecuado en los cinco

bits menos significativos del registro de estado. Para ello deben utilizarse las instrucciones de manipulación del registro de estado `mrs` y `msr`. La Tabla 2.2 resume su comportamiento.

Tabla 2.2: Instrucciones de manejo del Registro de Estado

Mnemotécnico	**Operación**
`MRS{<cond>} <Rd>, STReg`	`Rd <- STReg` donde `STReg` puede ser `CPSR` o `SPSR`.
`MSR{<cond>} STReg_campos, Op2`	donde los campos pueden ser `c,x,s,f` para los campos de control, extensión, estado y flags (ver figura 2.1), y `Op2` un registro o un inmediato. Modifica los campos indicados del registro de estado (`CPSR` o `SPSR`) con el valor de `Op2`.

Esta segunda alternativa es la que usaremos para inicializar los punteros de pila de todos los modos de ejecución. Por ejemplo, una posible secuencia de instrucciones para pasar a modo `Undef` e inicializar el puntero de pila del modo (si no lo estuviese previamente) 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 (ver Tabla 2.1) */
    msr     cpsr_cxsf,r1            /* Escribimos el resultado en el registro de
                                       estado, cambiando de éste los bits del
                                       campo de control, de extension, de estado y
                                       los de flag.  */
                    /* A partir de aquí estamos en modo Undef */
    ldr     sp,=UndefStack          /* Una vez en modo Undef copiamos la
                                       dirección de comienzo de la pila
(en esto consiste la inicialización de la pila)*/
```

2.2.2. Excepciones

La arquitectura ARM7TDMI (ARM V4T) 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** (`PAbort`) Cuando se realiza la búsqueda (fetch) de una instrucción en una dirección no válida. El controlador de memoria es el responsable de generar la excepción.
- **Data Abort** (`DAbort`) Cuando se intenta acceder a memoria en una posición no válida, para lectura o escritura de datos. Es el controlador de memoria el responsable de generar la excepción.

La tabla 2.3, ordenada de mayor a menor prioridad, muestra la correspondencia entre las excepciones, los modos de ejecución y los vectores. Observemos que cuando se inicializa el sistema (Reset) el modo de ejecución es SVC. es decir. el sistema arranca en modo supervisor, sin restricción alguna.

Tabla 2.3: 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`

Cuando se produce una excepción el procesador realiza automáticamente (por hardware) los siguientes pasos:

1. Almacena la dirección de retorno en el registro r14 propio del modo de ejecución asociado a la excepción. En realidad el valor almacenado depende del tipo de excepción[2] (consultar [arm]) lo que hace que el retorno de cada rutina de tratamiento de excepción sea distinto[3](ver la Tabla 2.4) , como veremos más adelante.

   ```
   R14_<modo_de_excepcion>  = Valor del PC cuando salto la excepción
   ```

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

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

[2]Cada excepción se detecta en una etapa distinta del procesador.
[3]Dependiendo de la excepción el retorno debe realizarse a la propia instrucción o la siguiente.

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[4].

   ```
   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 (ver Tabla 2.3).

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

Resumiendo, lo que sucede ante una excepción 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 2.3). Esta instrucción debe ser un salto a la rutina encargada de 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.

Rutinas de tratamiento de excepción

De la descripción anterior podemos deducir que una rutina de tratamiento de excepción debe preservar, como mínimo, el valor de los registros arquitectónicos `R0-R12`, ya que:

- No se modifica el registro `R14` del modo usuario, debido a que el registro de enlace utilizado es propio de cada modo de ejecución.
- Tampoco se modifica el registro `r13` (`SP`), que también es propio de cada modo.
- La preservación del `PC` se consigue escribiendo correctamente la dirección de retorno, que podemos obtener a partir de `R14_modo`.

[4] Las rutinas de tratamiento de excepción no pueden implementarse con el repertorio compacto *Thumb*.

Sin embargo, en nuestras prácticas optaremos por ser conservadores haciendo que las rutinas de tratamiento de excepción guarden en la pila el valor de todos los registros arquitectónicos.

Como cada modo tiene su propio registro de pila, es habitual que cada modo utilice un área de memoria distinto para la pila. Para que esto sea posible es necesario inicializar los registros de pila de cada modo con una dirección distinta.

Hay que tener en cuenta que, para regresar desde una rutina de tratamiento de excepción al punto donde se había interrumpido la ejecución del programa, hay que hacer **simultáneamente** (de lo contrario el retorno no sería correcto) dos cosas:

- restaurar el valor del `CPSR` a partir del valor guardado en el `SPSR`.
- escribir en `PC` la dirección de retorno, que podemos calcular a partir del valor almacenado en `LR` siguiendo las indicaciones de la tabla 2.4. Observemos que el cálculo concreto depende de la excepción.

Tabla 2.4: 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`

Hay dos mecanismos válidos para realizar correctamente el retorno:

- Se realiza el retorno mediante una instrucción de procesamiento de datos con el bit `S` activo (modificación del registro de estado) y empleando como registro destino `PC`[5] (ej. `SUBS PC, LR`).
- Se realiza el retorno mediante mediante una instrucción de load múltiple (`LDM`) con el bit `S` activo (modificación del registro de estado) y empleando `PC` como uno de los registros destino[5]. Es preciso señalar que para `LDM` la activación de `S` se lleva a cabo poniendo un acento circunflejo al final de la instrucción (ej. `LDMDB FP, {R0-R13, PC}^`).

El cuadro 1 muestra una posible estructura para una rutina de tratamiento de excepción por la línea IRQ de acuerdo con la primera alternativa.

Cuestión ¿Qué cambiaría respecto al código del Cuadro 1, el tratamiento la excepción DataAbort?

[5]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`

Cuadro 1 Rutina de tratamiento de IRQ para retorno `subs`

```
/* prólogo */
push  {r0-r10,fp}             @ Basta con apilar los registros modificados
                              @ INCLUYENDO r0-r3  si se modifican
add fp,sp,#(4*NumRegistrosApilados-4)

/* cuerpo de la rutina */

/* epílogo */
sub sp,fp, #(4*NumRegistrosApilados-4)
pop    {r0-r10, fp}    @ restauramos contexto y retornamos
subs pc,lr,#4       @ La constante a restar depende de la excepción
```

Escritura de rutinas de tratamiento de excepciones en C

Si queremos implementar las rutinas de tratamiento de excepción 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`.

Procediendo de esta forma gcc creará un rutina con una estructura similar a la descrita por el cuadro 1, en lugar de utilizar el prólogo y el epílogo de una función C.

2.3. Gestión de excepciones en la placa S3CEV40

Como hemos mencionado anteriormente, 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 2.3). 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 2.3 que los vectores corresponden a direcciones del comienzo del mapa de memoria, situadas en la ROM Flash. Dicha memoria contiene un programa de test suministrado por el fabricante, que comienza con un código similar al descrito en el cuadro 2, dónde la macro `HANDLER` y los símbolos utilizados son los del cuadro 3.

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 dada por la etiqueta `ResetHandler`, donde comienza la rutina encargada de gestionar la excepción *Reset*. Podemos comprobar que esta rutina se encuentra ubicada en la misma ROM Flash, por lo que no podremos sustituir la rutina de tratamiento de *Reset* sin reprogramar la Flash.

El resto de excepciones son tratadas de otro modo. Como ejemplo vamos a analizar en detalle lo que sucede en el caso de que se produzca una excepción *Undef*. Para facilitar la comprensión del proceso, la figura 2.3 ilustra la estructura de memoria del sistema de

Cuadro 2 Estructura del programa ubicado en la memoria ROM Flash, en la dirección 0x00 del mapa de memoria.

```
/*Comienzo del programa en dirección 0x00*/
start:
    b ResetHandler          /* 0x00 : vector de reset    */
    b HandlerUndef          /* 0x04 : vector de Undef    */
    b HandlerSWI            /* 0x08 : vector SWI         */
    b HandlerPabort         /* 0x0C : vector de Pabort   */
    b HandlerDabort         /* 0x10 : vector de Dabort   */
    b .                     /* 0x14 : utilizado en ARMv6 */
    b HandlerIRQ            /* 0x18 : vector de IRQ      */
    b HandlerFIQ            /* 0x1C : vector de FIQ      */

    /*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 */
```

Cuadro 3 Macro `HANDLER` y símbolos utilizados por el programa de test ubicado en la ROM Flash

```
.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
```

laboratorio, mostrando con mayor detalle las partes que intervienen en el procesamiento de la excepción *Undef*.

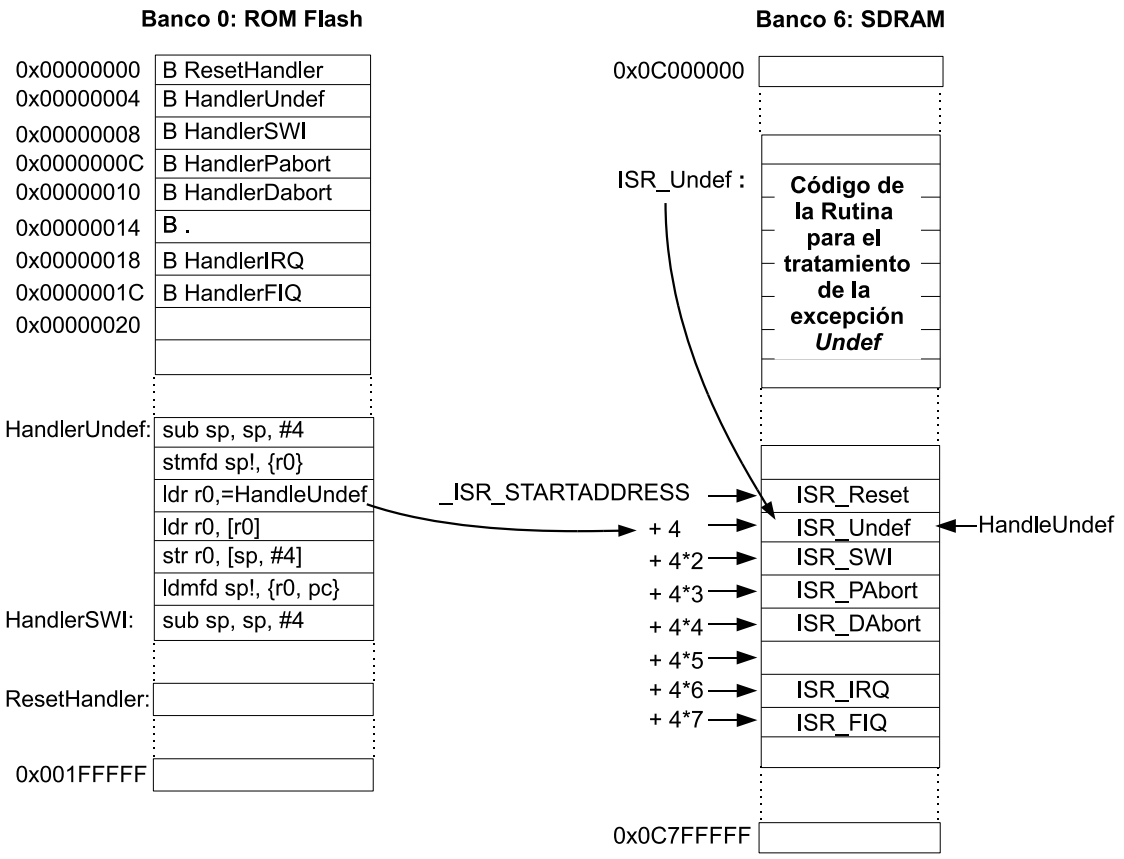

Figura 2.3: Datos e instrucciones involucrados en el procesamiento de una excepción *Undef*. El símbolo `_ISR_STARTADDRESS` indica el comienzo de la Tabla de Direcciones de ISRs y el símbolo `HandleUndef` indica la entrada de esta tabla correspondiente a la ISR de la excepción *Undef*.

Al producirse la excepción *Undef* el hardware pone en `PC` el valor `0x04` (vector de *Undef*), lo que produce un salto a la instrucción almacenada en esta dirección. Esta instrucción es un salto – relativo a `PC` – a la posición de memoria de la ROM Flash identificada por la etiqueta `HandlerUndef`, que marca el comienzo del fragmento de código encargado de obtener la dirección de comienzo de la ISR de *Undef* y realizar el salto a ella. Este fragmento de código ha sido generado mediante la macro `HANDLER`, descrita en el cuadro 3, del siguiente modo:

```
HandlerUndef HANDLER HandleUndef
```

Si analizamos este código (ver figura 2.3) observaremos que, después de salvar en la pila el registro `R0` para conservar su valor, lee de memoria el valor del símbolo `HandleUndef`, que se pasó como argumento a la macro, y lo guarda en `R0`. Este símbolo se corresponde con una dirección de la SDRAM que contiene a su vez la dirección de la rutina de tratamiento de la excepción *Undef* representada por el símbolo `ISR_Undef`. La instrucción `ldr r0,[r0]` se encarga de guardar esta dirección en el registro `R0` y la siguiente instrucción (`str r0, [sp, #4]`)

se encarga a su vez de guardarla en la pila. Por último, se restaura el valor de `R0` y se salta a `ISR_Undef` mediante la instrucción `ldmfd sp!,{r0,pc}`.

Resumiendo, para todas las excepciones salvo *Reset*, las direcciones de las rutinas encargadas de tratar las excepciones se leen de unas posiciones fijas de la memoria SDRAM, que habitualmente denominaremos *Tabla de Direcciones de ISRs*. Podemos ver además que estas posiciones de memoria comienzan en la dirección dada por el símbolo `\_ISR\_STARTADDRESS`, cuyo valor definido en el cuadro 3 es `0xc7fff00`.

Cuestiónes

- En el SoC del laboratorio, con la ROM tal y como la entrega el fabricante (por defecto), al recibir una excepción *Undef* se salta a la dirección 0x00000004. ¿Qué instrucción se encuentra en esa dirección? Dicho de otro modo, ¿en qué dirección de memoria comienza la rutina HANDLER para la excepción *UNDEF*?
- ¿En qué dirección está la entrada de la tabla que consulta la rutina HANDLER para la excepción *UNDEF* en la que encontraremos el comienzo de la ISR escrita por nosotros para tratar esa excepción? ¿Qué ocurre si no hemos inicializado esa posición de memoria y se produce esta excepción?

2.4. Mapa de memoria de un programa de 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 2.4. 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.

2.5. Entrada/salida mediante interrupciones

El procesador ARM7TDMI dispone de tres líneas que permiten interrumpir su ejecución de manera externa: una línea de RESET de comportamiento asíncrono, que genera una excepción de Reset, y dos líneas, IRQ y FIQ de comportamiento síncrono y enmascarables, que excepciones IRQ y FIQ respectivamente. Este tipo de excepciones generadas por elementos externos al procesador, suelen denominarse genéricamente interrupciones.

La excepción de RESET tiene por propósito la ejecución de un código que realiza la inicialización del sistema. Las excepciones de IRQ y FIQ, por su parte, tienen por propósito ejecutar el código necesario para atender al dispositivo que ha solicitado la interrupción. Ambas juegan por lo tanto un papel esencial en la gestión de la entrada/salida.

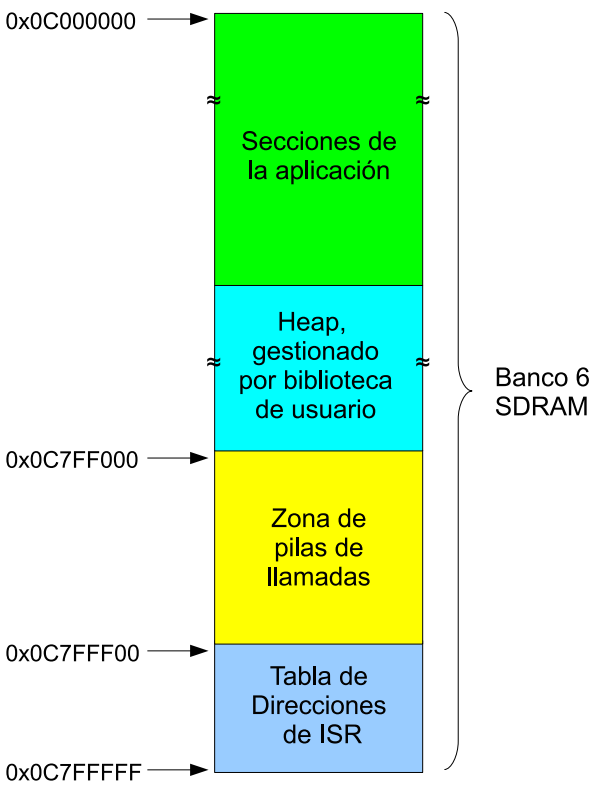

Figura 2.4: Mapa de memoria utilizado para los programas de las prácticas.

Las líneas IRQ y FIQ pueden enmascararse mediante dos bits del registro de estado (CPSR), I y F respectivamente (bits 6 y 7 según figura 2.1). Cuando estos bits toman el valor `'1'` las solicitudes de interrupción efectuadas por su correspondiente línea 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.

Identiticación de la fuente de interrupción

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. Como vimos anteriormente, en esta dirección de memoria debe almacenarse una instrucción de salto a la rutina de tratamiento correspondiente a la excepción.

Si varios dispositivos comparten la línea de petición de interrupción, ya sea IRQ o FIQ, generarán el mismo tipo de excepción y ejecutarán la misma rutina de tratamiento. En este caso, por lo tanto, es necesario que esta rutina (ISR_IRQ) identifique cuál ha sido el dispositivo que ha solicitado la interrupción. Para hacerlo, debe leer los registros de estado de los controladores de E/S asociados para comprobar si tienen una interrupción pendiente. A este proceso se le denomina *identificación por encuesta*. Asimismo, si hay varios dispositivos que hayan solicitado una interrupción simultáneamente, es la propia rutina la que tiene que decidir a cuál de ellos se atiende primero. Se dice en este caso que el *arbitraje es software*,

y habitualmente la prioridad viene dada por el orden en el que se consultan los registros de estado.

2.5.1. El controlador de interrupciones

La función de un controlador de interrupciones es la de mejorar/ampliar la gestión de interrupciones del procesador, en nuestro caso del ARM7TDMI. Para ello ha de situarse entre el procesador y los controladores de E/S encargados de gestionar los dispositivos, como podemos apreciar en la figura 2.5.

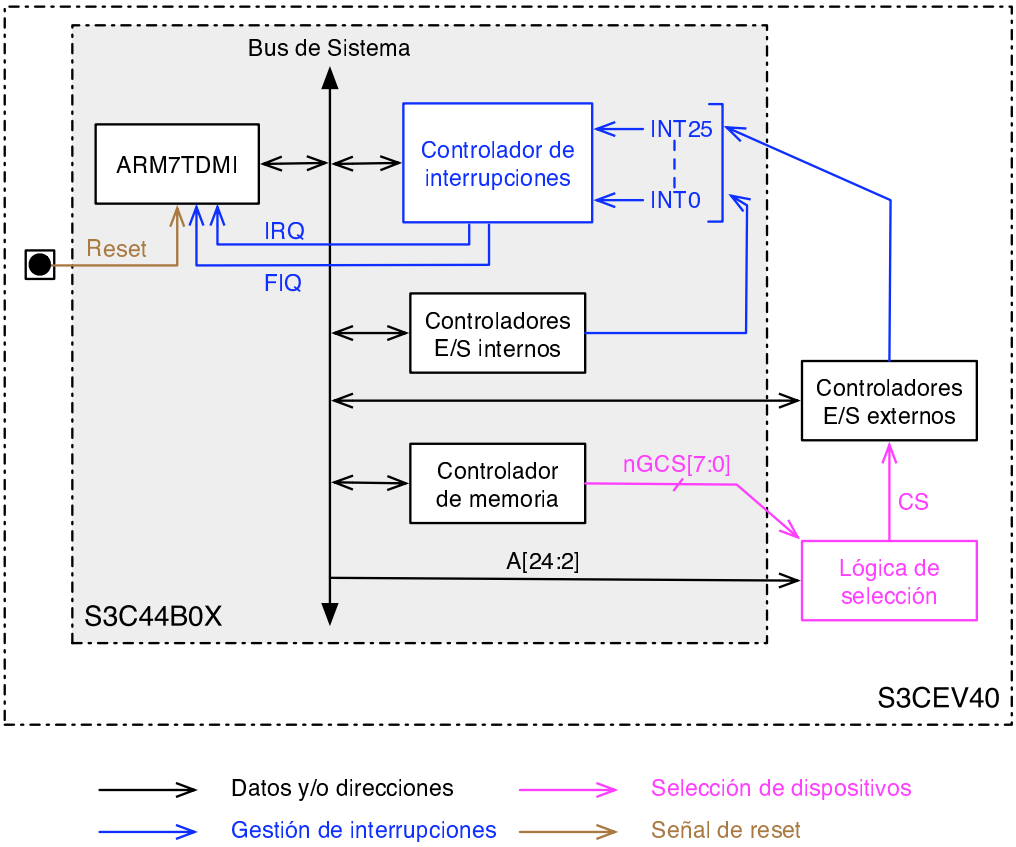

Figura 2.5: Sistema de E/S de la placa de laboratorio.

Aumento de líneas de interrupción

El controlador de interrupciones incluido en el Samsung S3C44B0X permite desdoblar las dos líneas de interrupción, IRQ y FIQ, en 26 líneas independientes enmascarables que pueden ser configuradas de forma que activen una de estas dos líneas, como veremos posteriormente. La figura 2.5 muestra de forma esquemática las principales señales que emplea el controlador. La tabla 2.5 muestra, entre otras cosas, el nombre que recibe cada una de estas líneas y el tipo de fuente de interrupción (controlador de DMA, UART, etc.).

Mejora del proceso de identificación

Este controlador permite además mejorar el proceso de identificación de la fuente de interrupción de dos formas distintas según el modo en el que haya sido configurado:

- *Modo No Vectorizado.*
  En este modo cuando una o más líneas del controlador se activan, el controlador activa – siempre que no estén enmascaradas – la línea FIQ o IRQ si alguna de ellas la tiene asociada. Recordemos que entonces el procesador ARM7TDMI generará el vector correspondiente a FIQ o IRQ según el caso[6], y saltará a la rutina programada para ese tipo de excepción. Para llevar a cabo la identificación, esta rutina debe consultar un registro del controlador de interrupciones que le indica cuáles son las líneas con interrupciones pendientes (consultando el registro EXTINTPND). Es preciso resaltar que este proceso de identificación es mucho más rápido que sin la presencia del controlador de interrupciones, ya que el procesador sólo tiene que consultar un registro del propio controlador, en lugar de consultar uno para cada dispositivo conectado a la línea (IRQ o FIQ). Por supuesto, si alguna de las líneas del controlador es compartida por varios dispositivos, para llevar a cabo la identificación es preciso consultar sus registros cuando la línea se active.
- *Modo Vectorizado.*
  En este modo cuando una o más líneas del controlador se activan:
  - Si alguna línea activa tiene asociada la interrupción FIQ, entonces se comporta como en el modo no vectorizado.
  - Si todas las líneas activas tienen asociada la interrupción IRQ, se producirá un excepción IRQ y se realizará un salto a la dirección 0x18 – vector de IRQ –, ubicación en la que está almacenada la instrucción encargada de saltar a la ISR de IRQ. No obstante, cuando el procesador pone en el bus esta dirección para leer esta instrucción, el controlador de interrupciones inserta en el bus otra instrucción de salto alternativa, evitando que se acceda a la que está almacenada en memoria. Este mecanismo, permite al controlador redirigir la ejecución de código a una posición de memoria distinta en función de la línea que se haya activado. El efecto conseguido con esto es como si cada línea del controlador estuviese autovectorizada. Los vectores generados para cada línea del controlador están recogidos en la tabla 2.5. Es conveniente señalar que en este modo, puesto que la identificación de la línea la hace el propio controlador, el abitraje (prioridad) debe también realizarlo él. Por defecto, cuando hay varias líneas activas – no enmascaradas – el controlador de interrupciones del S3C44B0X atiende en primer lugar a aquella línea cuyo número de línea sea mayor (EINT0 es la línea de mayor prioridad y ADC la de menor).

2.6. Manejo del controlador de interrupciones

La configuración y uso del controlador de interrupciones se hace a través de una serie de registros internos del controlador de interrupciones. Muchos de ellos tienen un bit por línea

[6] Si ambas líneas estuviesen activas simultáneamente, se atendería primero a FIQ ya su excepción es más prioritaria.

Tabla 2.5: Descripción de las 26 líneas de interrupción gestionadas por el controlador de interrupciones (externas al procesador). 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.

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

de interrupción, siguiendo la asignación bit-línea de la tabla 2.5. La relación de registros del controlador de interrupciones que necesitamos para la práctica se recoge en la tabla tabla 2.6, y su descripción es la siguiente:

INTCON *Interrupt Control Register* (`0x01E00000`). Registro de cuatro bits en el que sólo se utilizan tres de ellos:

- 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* (`0x01E00008`). Registro con un bit por línea: a '0' para que active IRQ o a '1' para que active FIQ.

Tabla 2.6: Registros del controlador de interrupciones

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

INTPND *Interrupt Pending Register* (`0x01E00004`). 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* (`0x01E0000C`). 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* (`0x01E00020`). 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). Es esencial cuando se están en *Modo Vectorizado* ya que el arbitraje es hardware.

I_ISPC *IRQ Int. Service Pending Clear register* (`0x01E00024`). 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 (en otras arquitecturas esta acción se conoce mediante *End of Interrupt* o simplemente EIO).

F_ISPC *FIQ Int. Service pending Clear register* (`0x01E0003C`). Igual que I_ISPC pero para la línea FIQ.

Para más información sobre estos registros es preciso consultar el manual de referencia del S3C44B0X [um-].

2.7. 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 primeros apartados sobre inicialización de excepciones, interrupciones no vectorizadas.

En la segunda se utilizarán interrupciones en modo vectorizado.

2.7.1. Primera parte: gestión de excepciones e interrupciones no vectorizadas

El objetivo de esta primera parte será familiarizarse con el funcionamiento de los modos de ejecución, las excepciones y las interrupciones no vectorizadas. Para ello el alumno deberá completar tres tareas:

- Inicializar correctamente el registro `SP` de cada uno de los modos de ejecución del procesador.
- Indicar qué rutinas de tratamiento de excepción/interrupción deben invocarse cuando se reciba cada una de las excepciones posibles en este ARM.
- Realizar el control de los pulsadores mediante interrupciones no vectorizadas. La pulsación de un botón supondrá el apagadao/encendido de un led

El programa comenzará con un código en ensamblador que inicialice el sistema y configure la tabla de direcciones de las rutinas de tratamiento de excepciones. Al finalizar invocará a la función *main*, que generará tres excepciones (Undef, Dabort y SWI), para posteriormente permanecer indefinidamente en un bucle.

El sistema interrumpirá la ejecución de ese bucle cuando se produzca la interrupción asociada a la pulsación de uno de los botones. Cada uno de los botones tendrá asociado un *led* que cambiará su estado (apagado/encendido) a cada pulsación del botón.

Se proporcionan los siguientes ficheros:

- `init.asm` En este fichero se deberán completar las dos primeras tareas anteriores. Se trata de una inicialización del sistema más completa que la efectuada en la práctica 1. En concreto, **se pide completar el código de las rutinas** `InitStacks` y `InitISR`. La figura 2.6 muestra un esquema de qué deben hacer ambas rutinas. Además, debe completarse la función `ISR_IRQ()` como se explica más abajo. Las rutinas más relevantes del fichero `init.asm` son:
  - `DoSWI, DoUndef, DoDabort`. Funciones que producen excepciones para estudiar el comportamiento del procesador al recibir una de ellas. Estas rutinas **se dan realizadas y el alumno NO debe modificarlas**.
  - `InitStacks`: Se encarga de inicializar las pilas de los distintos modos de ejecución, excepto el de usuario-system. Para ello, va cambiando de modo de ejecución, manteniendo enmascaradas las interrupciones, y en cada modo escribe en el registro de pila la dirección de comienzo correspondiente al modo, que está dada por el símbolo `<Modo>Stack`. Debemos recalcar que, a la salida de esta rutina el modo

debe ser privilegiado. Es más, como la subrutina se invocó desde `SVC` es preciso regresar a este modo para poder realizar correctamente el retorno de subrutina.

- `InitISR`: Se encarga de rellenar correctamente la Tabla de Direcciones de ISRs, ubicada en la SDRAM a partir del símbolo `_ISR_STARTADDRESS`. En concreto, para cada excepción deberá cargar en un registro la dirección de comienzo de la rutina que debe tratar la excepción, dada por el símbolo `ISR_<Excepción>`. Posteriormente cargará en un segundo registro la dirección de la SDRAM donde el código cargado en la ROM Flash espera encontrar la dirección de la rutina, dada por el símbolo `Handle<Excepción>`. Finalmente escribirá el contenido del primer registro (dirección de la ISR) en la dirección dada por el segundo registro (entrada correspondiente en la tabla de ISRs).

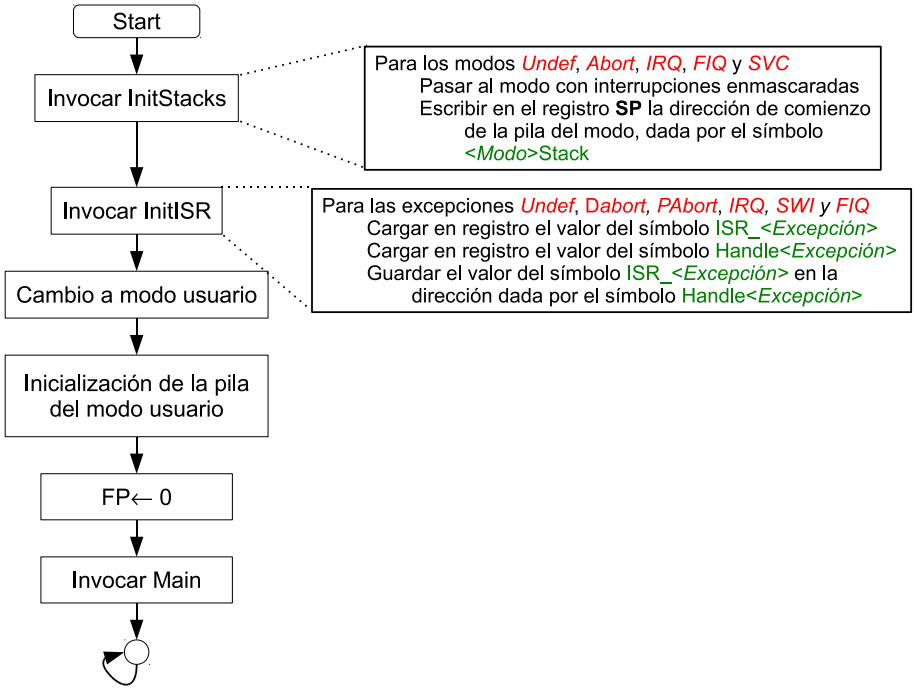

Figura 2.6: Diagrama de flujo del código de `p2a.asm`.

- `ISR_IRQ()`: **el alumno deberá implementar** (en ensamblador) esta rutina (rutina de tratamiento de la interrupción IRQ) Como las interrupciones se configurarán en modo no vectorizado, cualquier interrupción (timer, botones, teclado...) conlleva la ejecución de esta rutina. En ella, el alumno debe determinar el origen de la interrupción consultando al controlador de interrupciones y, una vez detectado el dispositivo que interrumpió, saltar a una función que trate dicha interrupción. Como en este caso sólo esperamos interrupciones de los botones, bastará con comprobar si son el origen de la interrupción y, si es así, saltar a la función `DoDetecta()`

Cuestión ¿Qué registro del controlador de interrupciones se debe consultar para determinar qué dispositivo produjo la interrupción en la línea IRQ?

- `boton.c`. Este fichero contendrá la función `DoDetecta()` que **el alumno deberá completar.** En esta función leeremos el registro `EXTINTPND` para determinar qué botón fue pulsado (y produjo la interrupción). Entonces, cambiaremos el estado del led asociado a ese botón.

- `led.c` Este fichero tendrá ya implementadas algunas funciones para encender y apagar cada uno de los *leds*. **El alumno deberá implementar las funciones** `switchLed1()` y `switchLed1()` que cambian el estado del *led* correspondiente (es decir, si está apagado lo enciende y si está encendido lo apaga).

- `main.c` Contiene una variedad de funciones, todas ellas ya implementadas **y no es necesario modificarlas (pero sí entenderlas)**. De entre ellas, destacamos:

  - `InitPorts()` Función que configura el puerto G y B para usar los botones y leds.
  - `IntInit()`. Función que inicializa el controlador de interrupciones, configurándolo en modo no vectorizado, que usaremos IRQ, activamos las líneas EINT4/5/6/7, activamos el modo en que la interrupción externa se dispara e inicializamos todas las interrupciones como *no pendientes*.
  - Rutinas de tratamiento de excepciones, excepto de IRQ (que deberá realizarla el alumno en el fichero `init.asm`). Todas estas rutinas escriben una cadena de caracteres con su nombre en una dirección de memoria específica (dada por la variable *screen*).

- Ficheros de cabecera que deben incluirse en el proyecto. De especial interés es el fichero `44b.h` en el que se definen numerosas macros para facilitar la lectura/escritura de los registros de dispositivo.

Cuestión 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.

2.7.2. Segunda parte: interrupciones vectorizadas

Esta segunda parte de la práctica consistirá en la implementación de un contador descendente que se controlará mediante los dos botones y cuyo valor se mostrará por el display 8-segmentos. Se utilizará un *timer* para llevar el ritmo de la cuenta. El sistema funcionará como sigue:

- El botón izquierdo será el botón de *stop/clear*. Siempre que se presione este botón, el contador se reseteará (es decir, tomará el valor `9`) y se quedará parado a la espera de comenzar la cuenta.

- El botón derecho será el botón de *play/pause*. Si el contador está parado, al pulsar este botón comenzará la cuenta atrás. Si el contador está en marcha, al pulsar este botón se parará en el número actual.
- Cuando el contador llegue a 0, se quedará en ese estado hasta que se pulse el botón de *stop/clear*.
- Cada interrupción del *timer* supondrá un decremento del valor del contador, siempre y cuando éste esté activo.
- Asímismo, los leds parpadearán con la frecuencia del *timer*.

Tanto los botones como el *timer* se gestionarán mediante **interrupciones vectorizadas**: deben tener su propia rutina de tratamiento de interrupción que se invocará de forma automática cuando pulsemos un botón o salte el *timer* (es decir, NO deberá ejecutarse la rutina de tratamiento de interrupción IRQ como se hacía en la parte anterior).

Todo el código que desarrollará el alumno será en C y se distribuirá en los siguientes ficheros:

1. `main.c` La función `main` se encargará de la invocar a las funciones de inicialización de los dispositivos que se vayan a utilizar. Posteriormente, permanecerá en un bucle infinito a la espera de interrupciones. Este código se entrega completo y **NO es necesario modificarlo.**
2. `boton.c`. Contiene todo el código relacionado con la gestión de los dos botones. Las funciones más relevantes son las de inicialización del dispositivo y la rutina de tratamiento de interrupción. Estas funciones se entregan parcialmente hechas, y el **alumno debe completarlas**.
3. `timer.c`. Contiene todo lo relacionado con la gestión del *timer*. Como en el caso anterior, las funciones de inicialización y la rutina de tratamiento de interrupción se entregan parcialmente desarrolladas, por lo que el alumno **deberá completarlas**.
4. `led.c`. Este fichero contiene la lógica de uso de los leds. Está completamente desarrollado y por tanto **no es necesario modificarlo**, si bien será necesario invocar alguna de sus funciones desde otros módulos.
5. `8seg.c`. En este fichero el alumno desarrollará el código para la representación del contador en el display 8 segmentos. **El fichero se entrega prácticamente vacío** para permitir al alumno desarrollar la interfaz que considere oportuna con el resto de módulos.
6. `44b.h` Fichero de cabecera con numerosas definiciones muy útiles para codificar nuestros programas. Tiene constantes para acceder a los registros (rPDATG, rPCONB, ...)de los dispositivos, del controlador de interrupciones (rINTMOD, rINTCON...) así como a la tabla donde deben registrarse las interrupciones (pISR_TIMER0, pISR_TIMER0). **Este fichero NO debe modificarse, pero sí consultarse**.
7. `44blib.c` Fichero con algunas funciones auxiliares como *Delay()* que permite realizar una espera de un tiempo determinado. Este fichero NO debe modificarse.

8. `44init.asm` Contienen código distribuido por le fabricante para inicialización del sistema. Este fichero NO debe modificarse.

Asimismo, el código distribuido en el campus también incluye otros ficheros de cabecera (y el script de enlazado) que deben incluirse en el proyecto Eclipse pero que NO deben modificarse.

Para ahorrar cálculos para determinar una frecuencia adecuada del *timer*, los valores de cuenta del generador de interrupciones del *timer* se entregan fijos. En concreto, como se ve en el fichero `timer.c`, los valores de los registros pertinentes son:

- rTCFG0 = 63;
- rTCFG1 = 0x0;
- rTCNTB0 = 65535;
- rTCMPB0 = 12800;
- rTCON = 0x2; /* establecer update=manual + inverter=on */
- rTCON = 0x09; /* inicial timer con auto-reload */

Cuestión Con esos valores en los registros de configuración del *timer*, ¿con qué frecuencia se producirá una interrupción de este dispositivo?

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.