

# *Práctica 5*

---

## *Introducción al sistema de Entrada/Salida*

---

### **5.1. Objetivos**

El objetivo de esta práctica es tener un primer contacto con los elementos fundamentales de un sistema de entrada/salida. Para ello utilizaremos la placa S3CEV40 descrita en clase, y únicamente haremos operaciones de E/S mediante encuesta. El estudio del sistema de interrupciones queda por tanto expresamente excluido de los objetivos de esta asignatura. Como ejemplo práctico usaremos los dispositivos más sencillos de los que disponemos: LEDs, pulsadores y el display de ocho segmentos. Los principales objetivos de la práctica son:

- Conocer el sistema de E/S de la placa S3CEV40 usada en el laboratorio
- Comprender el mecanismo de E/S localizado en memoria
- Aprender a programar dispositivos básicos mediante encuesta

### **5.2. Sistemas de Memoria y Entrada/Salida**

Los procesadores de ARM tienen *E/S localizada en memoria*, es decir, los registros de los controladores hardware de los dispositivos de entrada/salida tienen asignadas direcciones en el mismo espacio que la memoria. Como consecuencia, podemos leer o escribir en estos registros utilizando instrucciones `ldr`/`str`, o utilizar un puntero en C (ver la Sección 5.4).

Por motivos didácticos, hemos decidido dejar la descripción detallada del sistema en un documento adicional anexo, titulado *Sistemas de Memoria y Entrada/Salida de la placa S3CEV40*, que podrá ser utilizado como una documentación de referencia más. En él se explica el funcionamiento de los controladores de los dispositivos que vamos a utilizar en esta práctica, y se incluyen tablas que describen la funcionalidad de cada uno de los pines de los registros de estos controladores, así como las direcciones asignadas a dichos registros.

### **5.3. Uso de máscaras de bits**

A la hora de escribir/leer los registros de control/datos de un dispositivo, a menudo querremos consultar el valor de un bit concreto. Sin embargo, como ya hemos visto en numerosas ocasiones, los registros del ARM son de 32 bits. ¿Cómo podemos consultar el valor de un bit cualquiera de entre esos 32? ¿Cómo podemos ponerlo a 0/1 sin modificar

el resto? Lo mismo sucede en C si queremos modificar o consultar el valor de un sólo bit en una variable entera, que puede representar el valor que queremos escribir en un puerto o que hemos leído de él. Para conseguirlo, deberemos utilizar operaciones AND, OR, NOT y XOR a nivel de bit, bien sea en ensamblador o en C, utilizando un operando constante que denominaremos *máscara*.

### 5.3.1. Operaciones con máscaras en Ensamblador

Las operaciones que solemos considerar son:

- Consulta de un bit en un registro, generalmente porque queremos hacer unas acciones u otras en función de su valor (saltos condicionales). Para ello hacemos una AND del registro con una máscara que tenga todos los bits a 0 excepto el bit que queremos consultar (que pondremos a 1). Si el resultado es distinto de cero es que el bit consultado estaba a 1 en el registro. Si el resultado es 0 es que estaba a 0. Por ejemplo, si queremos consultar el valor del bit 6 (empezando a numerar en 0) del registro r1, podríamos hacer lo siguiente:

```
and r2, r1, #0x00000040
cmp r2, #0
beq CODIGO_PARA_BIT_A_CERO
@ CODIGO PARA BIT A 1
```

Entonces r2 tomaría el valor 0x00000040 si el bit 6 estaba a 1 y 0x00000000 en caso contrario. Por lo tanto, si el resultado es distinto de cero el bit estaba a 1. Si por el contrario el resultado es 0 significa que el bit estaba a 0.

- Poner a 0 algunos bits de un registro sin modificar el resto. En este caso hacemos una AND del registro con una máscara que tenga a 0 los bits que queremos poner a 0 y a 1 los bits que no queremos modificar. Por ejemplo, si queremos escribir un 0 únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

```
and r3, r3, #0xFFFFFD7
```

Muchas veces resulta más sencillo construir la máscara negando (complemento a 1) la máscara contraria. Hay varias formas de hacerlo, por ejemplo con `mvn`:

```
mvn r2, #0x00000028
and r3, r3, r2
```

- Poner a 1 algunos bits de un registro sin modificar el resto. En este caso hacemos una OR del registro con una máscara que tenga a 1 los bits que queremos poner a 1 y a 0 los bits que no queremos modificar. Por ejemplo, si queremos escribir un 1 únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

```
orr r3, r3, #0x00000028
```

- Invertir el valor de algunos bits, dejando el resto inalterados. En este caso hacemos una XOR del registro con una máscara que tenga a 1 los bits que queremos invertir y a

0 los bits que no queremos modificar. Por ejemplo, si queremos invertir únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

```
eor r3, r3, #0x00000028
```

Finalmente conviene que nuestro código sea lo más sencillo posible de seguir. El uso de máscaras dificulta la legibilidad del código y suele inducir a errores que son difíciles de encontrar, ya que al ver la máscara nos cuesta distinguir cuales son los bits que estamos modificando y cuales no. Por este motivo se recomienda definir símbolos para las máscaras y construir las máscaras con operaciones lógicas de máscaras sencillas, obtenidas con desplazamientos a la izquierda de 0x1. Por ejemplo, si queremos dos máscaras, la primera con los bits 1 y 2 a uno y el resto a cero, y la segunda con los bits 3 y 7 a 0 y el resto a uno, podemos escribir:

```
@definimos simbolos para las máscaras
.equ MASK1, ((0x1 << 1) | (0x1 << 2))
.equ MASK2, ~((0x1 << 3) | (0x1 << 7))

...

@usamos las máscaras
mov R1, #MASK1
mov R2, #MASK2
```

El ensamblador traduce estas instrucciones por:

```
mov r1, #6
mvn r2, #136
```

### 5.3.2. Operaciones con máscaras en C

En C podemos hacer las mismas operaciones utilizando los operadores a nivel de bit (*bitwise operators*), que son:

- AND: operador `&` (no confundir con el operador lógico `&&` que se utiliza para evaluaciones condición, tomando 0 como falso y distinto de cero como verdadero).
- OR: operador `|` (no confundir con el operador lógico `||`).
- NOT: operador `~` (no confundir con el operador lógico `!`).
- XOR: operador `^`.
- Desplazamiento a la izquierda: operador `<<`.
- Desplazamiento a la derecha: operador `>>`.

Así, las operaciones habituales serían:

- Consulta de un bit en una variable. Para ello hacemos una AND de la variable con una máscara que tenga todos los bits a 0 excepto el bit que queremos consultar (que pondremos a 1). Observemos que si el bit estaba a 1 el resultado será distinto de cero,

que es considerado como verdadero en C en cualquier evaluación lógica, y si el bit estaba a 0 el resultado será 0, que es considerado falso en cualquier evaluación lógica. Por ejemplo, si queremos ejecutar un código sólo si el 6 de una variable A está a 1, podríamos hacer lo siguiente:

```
if (A & 0x40) {
    // Código a ejecutar si el
    // bit 6 de A está a 1
}
```

- Poner a 0 algunos bits de una variable sin modificar el resto. Como en ensamblador, hacemos una AND de la variable con una máscara que tenga a 0 los bits que queremos poner a 0 y a 1 los bits que no queremos modificar. Por ejemplo, si queremos escribir un 0 únicamente en los bits 3 y 5 de A, dejando el resto inalterados, podríamos hacer lo siguiente:

```
A = A & ~(0x28);
```

que puede escribirse también como:

```
A &= ~(0x28);
```

- Poner a 1 algunos bits de una variable sin modificar el resto. Como en ensamblador, hacemos una OR de la variable con una máscara que tenga a 1 los bits que queremos poner a 1 y a 0 los bits que no queremos modificar. Por ejemplo, si queremos escribir un 1 únicamente en los bits 3 y 5 de A, dejando el resto inalterados, podríamos hacer lo siguiente:

```
A = A | 0x28;
```

que puede escribirse también como:

```
A |= 0x28;
```

- Invertir el valor de algunos bits, dejando el resto inalterados. En este caso hacemos una XOR de la variable con una máscara que tenga a 1 los bits que queremos invertir y a 0 los bits que no queremos modificar. Por ejemplo, si queremos invertir únicamente en los bits 3 y 5 de A, dejando el resto inalterados, podríamos hacer lo siguiente:

```
A = A ^ 0x28;
```

que puede escribirse también como:

```
A ^= 0x28;
```

Por supuesto, igual que comentamos para el caso del lenguaje Ensamblador, conviene que nuestro código sea lo más legible posible. En C suele ser habitual definir macros del preprocesador para las máscaras, construyéndolas con operaciones lógicas sobre máscaras sencillas obtenidas de desplazamientos a la izquierda de 0x1. Por ejemplo, si queremos dos

máscaras, la primera con los bits 1 y 2 a uno y el resto a cero, y la segunda con los bits 3 y 7 a 0 y el resto a uno, podemos escribir:

```
// definimos simbolos para las máscaras
#define MASK1 ((0x1 << 1) | (0x1 << 2))
#define MASK2 ~((0x1 << 3) | (0x1 << 7))

...
// Si los bits 1 y 2 están a 1
if ((A & MASK1) == MASK1) {
    //poner los bits 3 y 7 a 0
    A = A & MASK2;
}
```

## 5.4. Entrada/Salida en C

Como hemos venido explicando a lo largo de la práctica, un programa dedicado a atender dispositivos de E/S tendrá que leer y escribir en los registros de los controladores. Como en nuestra plataforma destino la E/S está localizada en memoria, cada puerto (registro de E/S) tiene asignada una dirección de memoria. Por tanto para leer del registro nos bastará con utilizar una instrucción `ldr` y para escribir en él un `str`. ¿Quiere esto decir que sólo podemos hacer la E/S en Ensamblador? Por fortuna la respuesta es no, gracias a que C incluye un tipo de datos para manejar direcciones de memoria, el puntero.

### 5.4.1. Punteros

Un puntero no es más que una variable que almacena una dirección de memoria. Podríamos pensar que esto no es nada nuevo, ya que cualquier entero de 32 bits puede ser utilizado para almacenar una dirección de 32 bits ¿verdad? Lo que pasa es que C proporciona además dos operadores unarios nuevos asociados a los punteros, que son:

- El operando de desreferenciación `*`, que debe ponerse delante de la variable puntero (`*variable_puntero`). La desreferenciación de un puntero implica un acceso a la dirección de memoria almacenada en el puntero (no hay que confundirla con la dirección en la que se almacena el propio puntero). Se suele decir que el puntero *apunta* a una dirección de memoria, y el operando `*` permite acceder a la dirección apuntada.
- El operando dirección-de `&`, que se puede poner delante de cualquier variable (`&variable_cualquiera`) para obtener la dirección en la que se almacena dicha variable. Este operando suele utilizarse para asignar a un puntero la dirección de una variable. Por ejemplo, esto es lo que se utiliza en C para pasar una variable por referencia a una función, en realidad se pasa por valor la dirección de la variable, que se asigna a una variable local de la función tipo puntero. Desreferenciando dicho puntero accedemos a la variable, y podremos modificar su valor dentro de la función.

Como vemos, se utilizan los mismos caracteres que para los operandos de multiplicación y AND bit a bit. Cuando un `*` aparece delante de una variable tipo puntero el compilador lo interpreta como una desreferenciación del puntero, y no como un operando de multiplicación.

Si el \* aparece delante de una variable entera, el compilador interpretará que es una operación de multiplicación, y no una desreferenciaciόn. Por este motivo necesitamos el tipo puntero. Si el resultado del operando dirección-de se asigna a una variable que no es de tipo puntero se producirá un error de compilaciόn.

En C los punteros son tipados, es decir, debemos identificar el tipo de la variable a la que apuntará el puntero. Para declarar un puntero a un tipo ponemos:

```
tipo * nombre_del_puntero;
```

Como cualquier otra variable, podemos declararla con valor inicial o sin él. Por ejemplo para declarar un puntero `mipuntero` que sirva para apuntar a enteros escribiríamos:

```
int *mipuntero;
```

¿Y qué quiere decir que al desreferenciar un puntero se accede a la dirección apuntada por dicho puntero? La respuesta exacta a esta pregunta depende de la arquitectura para la que se genere el código. En el caso del ARM (y cualquier otro procesador RISC) el acceso implica:

- Un `ldr` a la dirección almacenada en el puntero (la dirección donde apunta) si la desreferenciación implica una lectura. Por ejemplo si aparece a la derecha de un igual, en un paso como parámetro a una función, en la condición de un `if`, etc.
- Un `str` a la dirección almacenada en el puntero si la desreferenciación implica una escritura, es decir, a la izquierda de un igual en una asignación.

Como los punteros son tipados, la operación `ldr/str` será la adecuada para el tamaño del dato apuntado. Por ejemplo, si es un puntero `char*` se utilizarían las instrucciones `ldrb/strb`.

Entonces, para poder realizar la E/S en lenguaje C no tenemos más que asignarle a un puntero la dirección del registro de E/S que queramos acceder y desreferenciar el puntero. Por ejemplo, supongamos que queremos poner a 0 el bit 6 del registro PDATB para encender el led conectado al pin 6 del puerto B. En ensamblador haríamos lo siguiente:

```
ldr r0,=0x1d2000c @ Cargamos en r0 la dirección de PDATB
ldr r1, [r0]
and r1, r1, #~(0x1<<6)
str r1, [r0]
```

y en cambio en C haríamos:

```
// declaramos un puntero rPDATB y lo inicializamos
unsigned int* rPDATB = (unsigned int*) 0x1d2000c;
// lo usamos para leer el registro
// y volver a escribir un nuevo valor
*rPDATB = *rPDATB & ~(0x1 << 6);
```

Además de la desreferenciación, C permite sumar o restar un valor entero a un puntero. Ojo, no permite sumar dos punteros, sino sumar un valor entero a un puntero. Como programadores de ensamblador debemos estar acostumbrados a esta operación, el puntero representa la dirección base y el entero el desplazamiento, como en el caso de las instrucciones `ldr/str`. Sin embargo hay un matiz importante. Gracias a que los punteros son tipados, el compilador sabe cuántos bytes ocupa el tipo de dato apuntado por el puntero. Cuando

sumamos un entero *i* a un puntero de un tipo que ocupa *n* bytes, el código generado por el compilador suma *i\*n* a la dirección almacenada en el puntero. Esto permite utilizar cómodamente un puntero para recorrer un array. Si inicializamos el puntero con la dirección del primer elemento del array y le sumamos *i* al puntero, obtendremos la dirección del elemento con índice *i*. Por ejemplo, en el siguiente código utilizamos un puntero para escribir el valor 10 en la posición 5 del array A (es decir en A[5]):

```
unsigned int A[100] ={0};
unsigned int *p = A;
*(p + 5) = 10;
```

Finalmente, podemos también indexar un puntero como si fuese un array. Es decir, si *p* es un puntero, podemos poner *p[i]* para acceder a la dirección almacenada más *i* veces el número de bytes ocupados por el tipo apuntado, es decir, es equivalente a poner *\*(p+i)*.

#### 5.4.2. Direcciones volátiles

Prácticamente sabemos ya todo lo que hay que saber para poder manejar los puertos de E/S en lenguaje C, tan sólo falta darnos cuenta de una importante sutilidad: el contenido de los registros de E/S puede cambiar sin que lo cambie el programa, por ejemplo, porque pulsamos un pulsador. Esto no lo sabe el compilador si no se lo indicamos. Para ilustrar el problema supongamos que tenemos un bucle que está examinando el estado de los pulsadores conectados a los pines 6 y 7 del puerto G, esperando a que se pulse uno de ellos. El código podría ser el siguiente (para comprender los detalles del código deben consultarse las Secciones 2 y 3 del documento adicional anexo, titulado *Sistemas de Memoria y Entrada/Salida de la placa S3CEV40*) :

```
#define BUTTONS (0x3 << 6)
unsigned int *rPDATG = (unsigned int*) 0x1d20044;
unsigned int status;

...
do {
    status = ~(*rPDATG);
    status = status & BUTTONS;
} while (status == 0);
```

y esperaríamos que el compilador tradujese este código por uno similar a este:

```
.equ rPDATG, 0x1d20044
.text
...
ldr r0, =rPDATG
D0:
    ldr r1,[r0]
    mvn r1, r1
    and r1, r1, #(0x3 << 6)
    cmp r1, #0
    beq D0
```

Fijémonos que dentro del bucle no se modifica lo que hay en la dirección 0x1d20044, sólo se lee. El compilador podría entonces *optimizar* el código sacando el `ldr` fuera del bucle, con lo que quedaría:

```
.equ rPDATG, 0x1d20044

.text
...
ldr r0, =rPDATG
ldr r1,[r0]
DO:
    mvn r1, r1
    and r1, r1, #(0x3 << 6)
    cmp r1, #0
    beq DO
```

Entonces si no hay ningún pulsador pulsado cuando se entra en el bucle, el bucle será infinito. ¿Qué ha pasado? El compilador asume que si el código que él genera no cambia el valor almacenado en 0x1d20044 le basta con leerlo una vez al principio (¡porque no va a cambiar!). Lo que hay que hacer es indicarle al compilador que esa dirección de memoria es volátil, que su valor puede cambiar por mecanismos ajenos al programa, y que por tanto debe volver a leer el valor cada vez que quiera consultarla. Esto se consigue utilizando el modificador `volatile` en la declaración del puntero. Así, el código correcto para escanear los pulsadores sería:

```
#define BUTTONS (0x3 << 6)
volatile unsigned int* rPDATG = (unsigned int*) 0x1d20044;
unsigned int status;

...
do {
    status = ~(*rPDATG);
    status = status & BUTTONS;
} while (status == 0);
```

En este caso el compilador nunca sacará el `ldr` fuera del bucle.

#### 5.4.3. Uso de macros para acceder a los puertos de E/S

Cuando se programa la E/S en C, es frecuente utilizar macros del preprocesador para el acceso a los puertos, en lugar de definir punteros. Esto permite al compilador generar un código más eficiente. ¿Pero no hemos dicho que necesitamos los punteros? Así es, pero C permite convertir un literal a un puntero mediante un *casting*. Esto podemos hacerlo directamente en una macro del preprocesador. Además incluyendo en la macro el símbolo `*` para la desreferenciar el puntero podemos obtener un código más legible. Por ejemplo, el código anterior para el escaneo de los pulsadores podría quedar así:

```
#define BUTTONS (0x3 << 6)
#define rPDATG (*(volatile unsigned int*) 0x1d20044)
```

```

unsigned int status;

...

do {
    status = ~(rPDATG);
    status = status & BUTTONS;
} while (status == 0);

```

En esta práctica incluiremos todas las macros para el acceso a los puertos B y G y al display de 8 segmentos en un fichero de cabecera `ports.h`. Bastará entonces con incluir este fichero (con la directiva `#include`) en los ficheros fuente en los que queramos trabajar con alguno de los puertos de E/S.

#### 5.4.4. Estructura de un programa dedicado a Entrada/Salida

Determinadas aplicaciones están enfocadas únicamente a realizar tareas de E/S: leer valores de sensores (temperatura, luz, posición), teclado u otros, y actuar sobre elementos externos al procesador ( motores, displays, leds) en función de la información obtenida en la lectura. En esos casos, la estructura del código de dichas aplicaciones, extremadamente simplificada y asumiendo una E/S basada en encuesta y espera activa, podría esquematizarse como sigue:

```

void main(void)
{
    // Configurar dispositivos como entrada y/o salida
    inicialización_de_dispositivos;

    while (true) {

        // Rutina(s) para leer los valores de cada dispositivo
        // A su vez, contendrán bucles de espera para el dispositivo
        lectura_dispositivos_de_entrada;

        calcular_salidas_en_función_de_etradas;

        escritura_dispositivos_de_salida;
    }
}

```

En el desarrollo de esta práctica seguiremos fielmente ese modelo.

### 5.5. Configuración de un proyecto para la placa S3CEV40

Para esta práctica añadiremos como de costumbre un nuevo proyecto a nuestro `workspace`. Toda la configuración de compilación se hace igual que siempre. Sin embargo, en esta ocasión queremos ejecutar nuestro código sobre el arm7tdmi de la placa S3CEV40, en lugar de utilizar el simulador, por lo que no nos vale la configuración de depuración que hemos venido utilizando.

Lo primero que debemos hacer es configurar Eclipse para utilizar una herramienta externa. Para ello seleccionamos `Run→External Tools→External Tools Configurations...`

(también puede llegarse al mismo sitio pinchando en la flecha del botón ). Se abrirá una ventana en la que podemos configurar el uso de una nueva herramienta externa. Para ello debemos pinchar en **Program**, con lo que se habilitarán unos botones y pinchamos en el primero de ellos (). La Figura 5.1 muestra cómo debemos llenar el resto de la ventana. Para llenar la primera entrada podemos pinchar en el botón **Browse File System...** y buscamos el ejecutable de OpenOCD en el sistema (téngase en cuenta que la ruta mostrada en la figura puede ser distinta a la ruta que tengamos que poner en el laboratorio). Del mismo modo, para seleccionar el directorio de trabajo podemos pinchar en el botón **Browse Workspace....**. Finalmente debemos llenar los argumentos al programa (-f test/arm-fdi-ucm.cfg) tal y como se indica en la Figura 5.1. Terminamos pinchando en **Apply** y **Close**. Esta herramienta externa la tendremos que ejecutar antes de comenzar la depuración, con la placa conectada a un puerto USB del equipo de laboratorio, tal y como indicamos en la siguiente sección.

Además de añadir OpenOCD como herramienta externa debemos crear una nueva configuración de depuración. Los pasos iniciales se dieron en la práctica 1. Lo único que cambia es que en esta ocasión en lugar de seleccionar **Zylin** debemos seleccionar **GDB Hardware Debugging**, como muestra la Figura 5.2. Como en las prácticas anteriores, damos un nombre a la configuración, seleccionamos el proyecto y el ejecutable. En este caso debemos además pinchar en un enlace azul que aparece en la base de la ventana que pone **Select Other...**. Se abrirá entonces una ventana como la mostrada en la Figura 5.3, en donde deberemos marcar la casilla **Use configuration specific settings** y seleccionar **Standard GDB Hardware Debugging Launcher**.

A continuación debemos seleccionar la pestaña **Debugger**, y rellenarla como indica la Figura 5.4. Primero deberemos seleccionar como depurador el **arm-none-eabi-gdb**. Después, en la parte baja de la ventana, deberemos seleccionar a qué *gdb server* debe conectarse. Este servidor lo pone OpenOCD, y está configurado para escuchar en el puerto 3333. Por ello seleccionaremos como **JTAG Device Generic TCP/IP**, como dirección IP **localhost** y como puerto 3333.

Finalmente deberemos llenar la pestaña **Startup**, tal y como se ilustra en la Figura 5.5. En la parte superior desmarcamos las casillas de **Reset** y **Halt**, y escribimos en el cuadro **monitor reset init**. En la parte inferior de la ventana marcamos las casillas **Set Breakpoint at** y **resume**, y ponemos **\*start** en la casilla para la dirección del breakpoint. Pinchamos en **Apply** y luego en **Close**. La configuración de depuración queda entonces lista para ser utilizada cuando queramos depurar.

## 5.6. Desarrollo de la Práctica

En esta práctica vamos a partir de un programa, proporcionado junto con este documento, que hace que los dos leds de la placa parpadeen aproximadamente a una frecuencia de 1Hz. El programa estará compuesto de varios ficheros fuente, la mayor parte de ellos implementan funciones para actuar sobre un dispositivo de la placa. Sin embargo, algunas de estas funciones no estarán implementadas. El alumno deberá completar la implementación de estas funciones. Una vez completadas, se propone al alumno una modificación del programa principal que utilice todas estas funciones. Los pasos a seguir son:

1. Creamos un proyecto de Eclipse y le añadimos los ficheros que acompañan a la práctica, que son:

- **ports.h**: definición de macros para el acceso a los puertos de E/S.

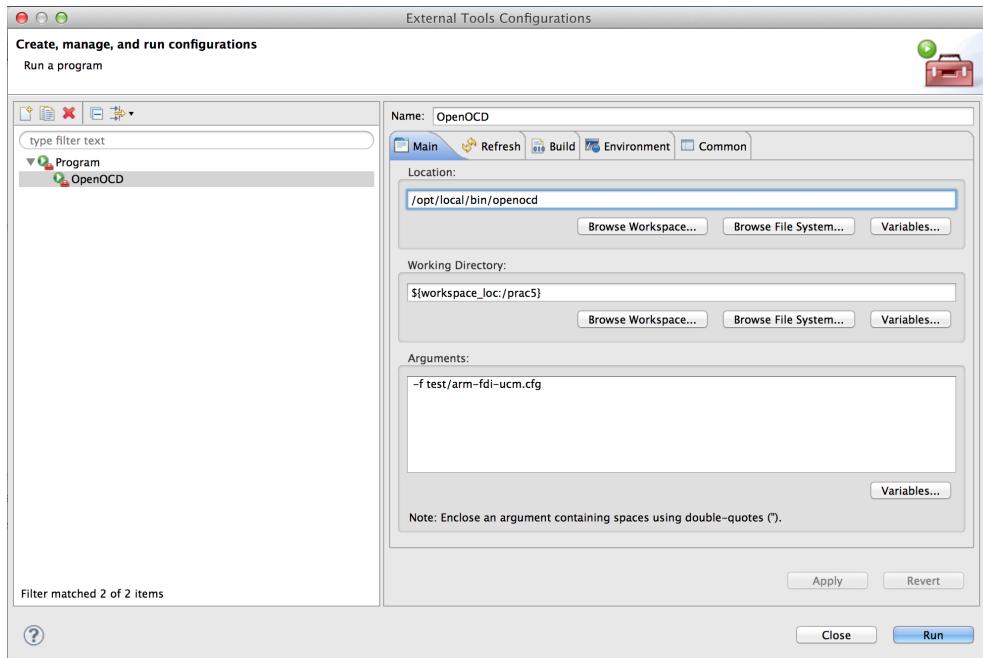


Figura 5.1: Ventana de configuración de OpenOCD como herramienta externa.

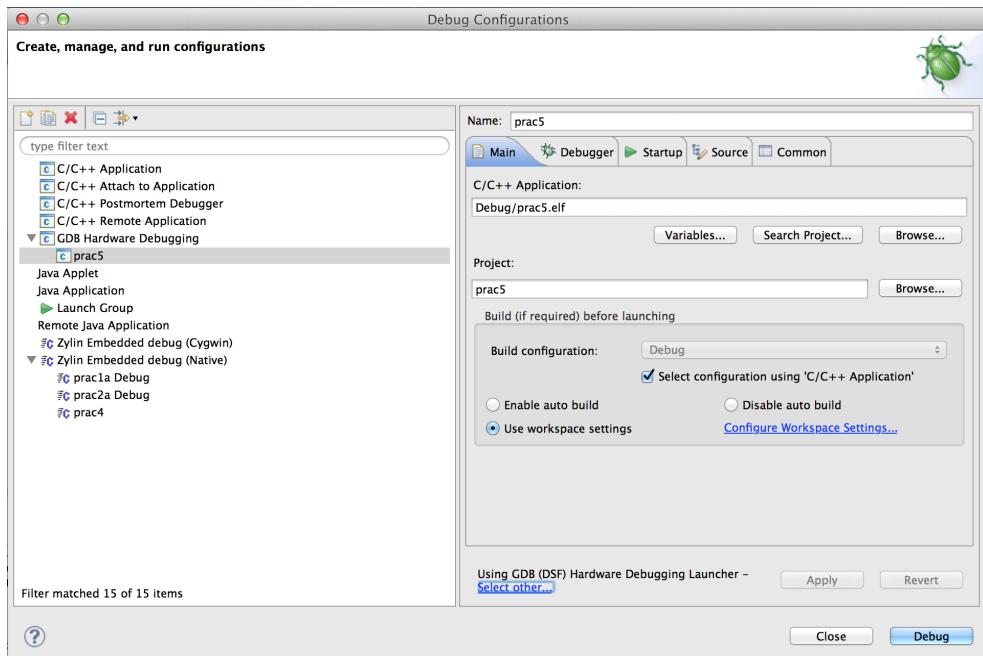


Figura 5.2: Configuración de depuración para conexión a placa: pestaña Main.

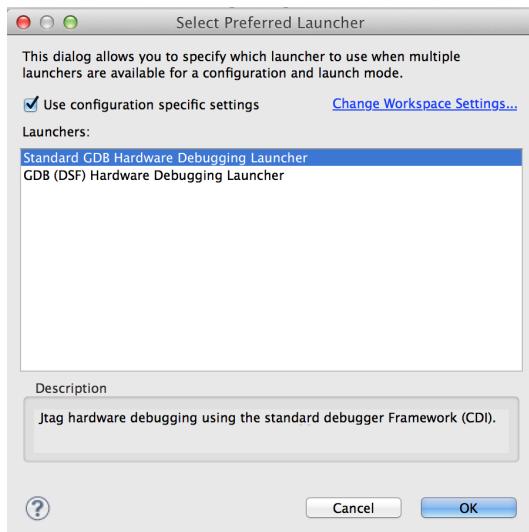


Figura 5.3: Configuración de depuración para conexión a placa: selección de *Standard GDB Hardware Debugging Launcher*.

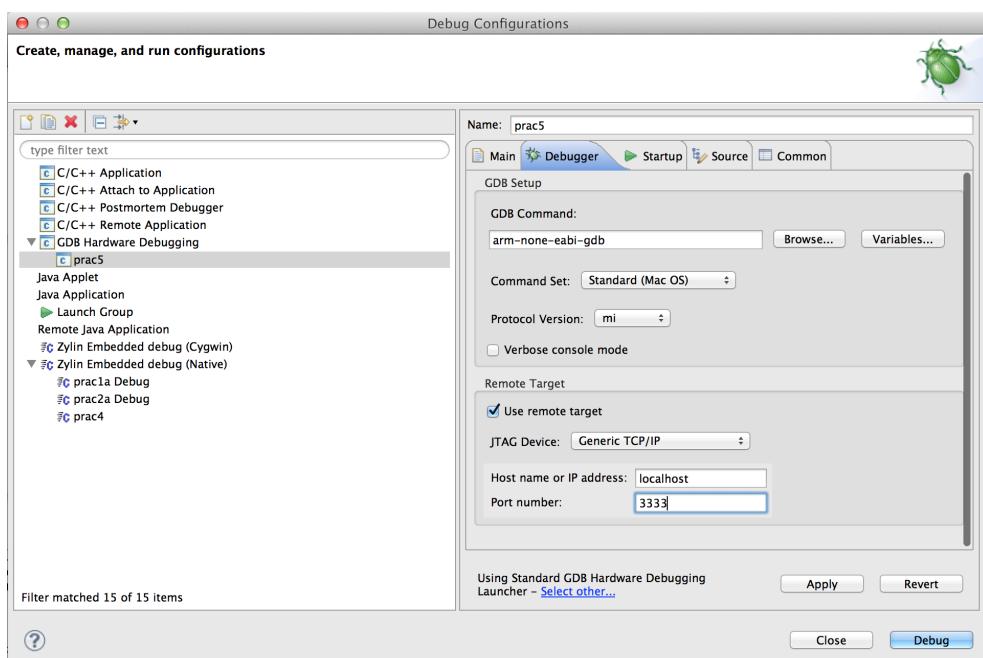


Figura 5.4: Configuración de depuración para conexión a placa: pestaña Debugger.

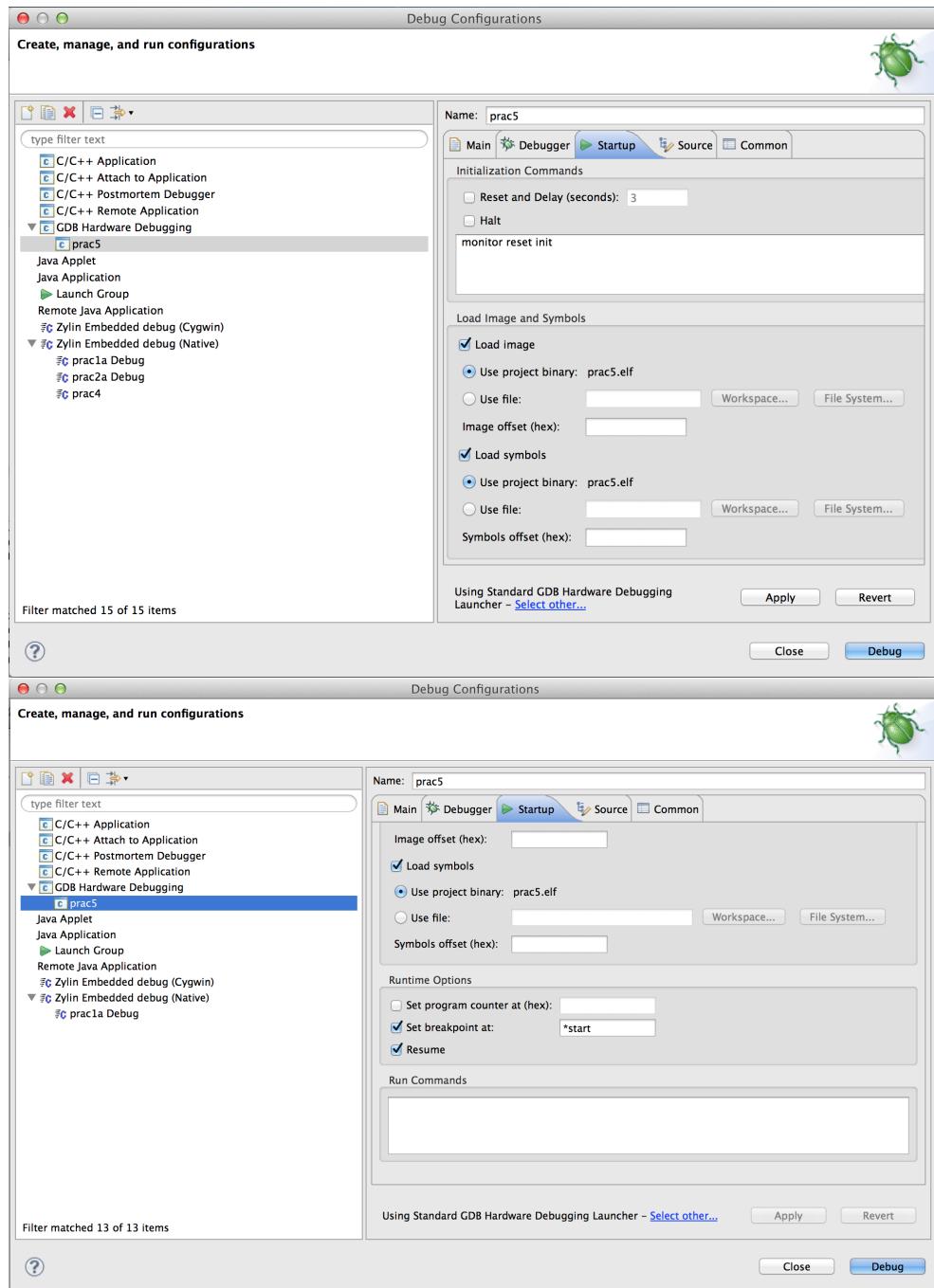


Figura 5.5: Configuración de depuración para conexión a placa: pestaña Startup.

- `button.h`: declaración de funciones para manejar los pulsadores.
  - `button.c`: implementación de las funciones para manejar los pulsadores.
  - `leds.h`: declaración de las funciones para manejar los leds.
  - `leds.c`: implementación de las funciones para manejar los leds.
  - `D8Led.h`: declaración de las funciones para manejar el display de 8 segmentos.
  - `D8Led.c`: implementación de las funciones para manejar el display de 8 segmentos.
  - `utils.h`: declaración de la función `Delay` que usaremos para realizar esperas activas.
  - `utils.c`: implementación de la función `Delay`.
  - `main.c`: programa principal
  - `init.asm`: programa de inicialización
  - `ld_script.ld`: script de enlazado
2. Configuramos la compilación del proyecto como en las prácticas anteriores y compilamos.
  3. Cambiamos a la perspectiva de Debug.
  4. Conectamos la placa a un puerto USB libre del equipo de laboratorio (PC).
  5. Ejecutamos la herramienta externa OpenOCD, creada previamente siguiendo los pasos descritos en la Sección 5.5.
  6. Ejecutamos la configuración de depuración creada previamente siguiendo los pasos descritos en la Sección 5.5.
  7. Si todo ha ido bien estaremos parados en el símbolo `start`, al inicio de nuestro programa. Dejamos correr el programa. Los leds deberían parpadear con una frecuencia de 1Hz, los dos a la vez.
  8. Para la evaluación de la práctica el profesor suministrará a los alumnos el código del guión completado, y el alumno deberá:
    - a) Analizar la solución, volcarla a la placa y comprobar el funcionamiento haciendo pequeñas pruebas con los dispositivos implicados.
    - b) Realizar un pequeño programa que haga algo con los dispositivos. Lo que haga el programa se deja a la libre elección de los alumnos, teniéndose en cuenta la originalidad y dificultad de lo que hagan en la nota final de la práctica. A modo de ejemplo se dan las siguientes ideas:
      - Programa haga que el display de 8 segmentos muestre un sólo segmento encendido, girando en sentido horario a una frecuencia de 1Hz. Cada vez que pulsemos el botón 1 el led cambia el sentido del giro. Si pulsamos el botón 2 el segmento se detiene. Si lo volvemos a pulsar reanuda su movimiento. Observese que en este caso debe cambiarse la función de lectura de los pulsadores, porque no podemos quedarnos bloqueados indefinidamente esperando a que el usuario pulse alguno. Esto podemos hacerlo añadiendo un parámetro a la función que nos diga si queremos esperar o no, haciendo sólo una iteración del

bucle si no queremos esperar. Una solución sencilla sería muestrear unas 10 veces por segundo, usando `Delay` para esperar 100ms después de comprobar el estado de los pulsadores, de forma que cada 10 muestreos (1Hz) hagamos el cambio de posición del segmento si está en movimiento y hacemos parpadear los leds. Esta espera además nos serviría para filtrar los rebotes (aunque podemos detectar varias pulsaciones seguidas si dejamos el dedo en el pulsador, ya que no esperamos a que el usuario deje de pulsar el botón).

- Programa que muestre cíclicamente y letra a letra la palabra `hola` en el display de 8 segmentos. El botón 1 permite alternar entre dos velocidades, de 1Hz y 2Hz (1 o 2 letras por segundo). El botón 2 permite alternar entre dos palabras `hola` y `adios`. Recordar que debe darse soporte en `D81ed.c` para mostrar los caracteres que necesitamos.