Práctica 10

Desarrollo de un driver L2 para las UART

Introducción

Con los *drivers* L1 desarrollados en la práctica anterior, ya podemos disfrutar de una E/S no bloqueante en nuestra plataforma. Sin embargo, para acceder a los dispositivos sigue siendo necesario conocer el API de sus *drivers* en nuestro BSP, lo que dificulta el desarrollo de aplicaciones. Para solucionar este inconveniente, en esta práctica se completará el BSP para dotar a los *drivers* de una capa L2, que permitirá acceder a los diferentes dispositivos de la placa mediante la interfaz de E/S estándar definida en la biblioteca estándar de *C*. Para ello, y dado que en nuestra *toolchain* se usa la biblioteca *Newlib* como implementación de la biblioteca *libC*, será necesario conocer la organización interna de *Newlib*, ya que debemos escribir el código de los *drivers* L2 de nuestro BSP de forma que se integre correctamente con esta biblioteca.

Una vez proporcionada esta capa L2, que es la que implementará los detalles específicos de la plataforma, se podrán usar sin problemas el resto de funciones estándar de *libC* para acceder a los dispositivos de nuestra placa, como la E/S basada en *streams* (`fopen`, `fclose`, etc.) para mejorar las prestaciones de las operaciones sobre los dispositivos, y la E/S formateada (`fprintf`, `fscanf`, etc.), para permitir una E/S más versátil a las aplicaciones.

Objetivos

- Entender la organización interna de la biblioteca *Newlib* para saber cómo debemos integrar nuestro BSP con ella.
- Entender el API de gestión de dispositivos y descriptores de fichero del BSP.
- Implementar las llamadas a sistema necesarias para completar el soporte de *drivers* L2 en el BSP.

La biblioteca Newlib

El lenguaje de programación *C* dispone de una serie de funciones estándar para la gestión de la memoria, entrada/salida, ficheros, cadenas de caracteres, implementación de funciones con un número variable de argumentos, etc. La declaración de estas funciones está separada en una serie de ficheros cabecera, como por ejemplo `stdlib.h`, `stdio.h`, `string.h`, etc., que deberemos incluir al principio de nuestra aplicación para que el compilador pueda comprobar que las llamadas a las funciones que usemos de la biblioteca sean correctas sintácticamente. Sin embargo, el código de todas las funciones está dentro de un fichero de biblioteca llamado `libc.a`, con el que tendremos que enlazar nuestra aplicación para poder construirla.

Además de la biblioteca de funciones estándar *libC*, el lenguaje *C* también dispone de otra biblioteca estándar de funciones matemáticas, la biblioteca *libM*, en la que podemos encontrar funciones para la generación de números aleatorios, funciones trigonométricas, exponenciales, raíces cuadradas, etc. Para hacer uso de estas funciones es necesario que incluyamos el fichero cabecera `math.h` al principio de nuestro programa, que lo compilemos, y que lo enlacemos con la

biblioteca de funciones.

La biblioteca *Newlib* es una implementación para sistemas empotrados de las bibliotecas *libC* y *libM*. Como sólo se distribuye su código fuente, es necesario construirla para la arquitectura específica de nuestro sistema empotrado (ya lo hicimos en el seminario 2). Una vez construida, para usarla simplemente hay que incluir en nuestra aplicación las cabeceras que declaren las funciones que queremos utilizar, y después hay que indicarle al enlazador que enlace con los ficheros `libc.a` y/o `libm.a`, dependiendo de las funciones con las que queramos enlazar nuestra aplicación, tal y como se muestra en los *Makefiles* proporcionados a lo largo de las prácticas.

Características adicionales

Además de ofrecer una implementación compacta y eficiente de las bibliotecas *libC* y *libM*, a continuación se describen otras características que hacen que *Newlib* sea una biblioteca muy apropiada para el desarrollo de sistemas empotrados[1].

La función iprintf

Además de proporcionar una implementación completa de todas las funciones de la familia de `printf`, *Newlib* también añade la función `iprintf`, que al no incluir soporte para formatear números en coma flotante (la mayoría de sistemas empotrados no los usan), se codifica con muchas menos líneas de código ensamblador, lo que favorece la disminución del tamaño final de nuestro ejecutable.

Versiones *float* de las funciones matemáticas

Además de las funciones estándar de la biblioteca *libM* (`sin`, `pow`, etc.), que operan con números representados en coma flotante de doble precisión, *Newlib* también incorpora versiones `float` de estas funciones (`sinf`, `powf`, etc.) que operan con números reales de precisión sencilla. Esta reducción de precisión supone una reducción de carga computacional en la aplicación, algo ideal en la mayoría de las aplicaciones empotradas.

Soporte multihebra

A la hora de construir la biblioteca se puede indicar si se desea que sus funciones tengan soporte para llamadas reentrantes. Por defecto se construye la biblioteca sin este soporte. Para habilitarlo, tan sólo hay modificar el *Makefile* para añadir la opción `-DREENTRANT_SYSCALLS_PROVIDED` a la variable `CFLAGS_FOR_TARGET`[2].

Facilidad de depuración

Cuando se construye la biblioteca *Newlib,* además del fichero `libc.a`, en el que se encuentran todas las funciones de la biblioteca *libC*, también se genera el fichero `libg.a`, que además de contener estas funciones, también contiene información para su depuración. Para usar `libg.a` en vez de `libc.a` simplemente hay que cambiar en nuestro *Makefile* la opción de enlazado `-lc` por `-lg`.

Portabilidad

La biblioteca *libC* de *Newlib* depende de un conjunto de funciones (`open`, `kill`, `fork`, etc.) que

1 B. Gatliff. Embedding with GNU: Newlib. *Embedded Systems Programming*, 15(1), 2002.
2 B. Gatliff. Embedding with GNU: Newlib part 2. *Embedded Systems Programming*, 15(1), 2002.

proporcionan las llamadas al sistema básicas que el resto de funciones de la biblioteca necesitan, como la gestión de memoria, procesos, ficheros, hora del sistema, etc. Para facilitar la portabilidad de la biblioteca, estas funciones simplemente realizan llamadas a funciones *stub*, que deben estar implementadas en el BSP de cada plataforma a la que se quiera portar *Newlib*, y que deben solucionar el acceso a bajo nivel a los recursos del sistema.

Normalmente, cuando se construye la biblioteca *Newlib,* se genera un fichero llamado `syscalls.c`, que se incluirá en la biblioteca *libC*, y que contendrá una implementación por defecto para las funciones *stub.* En el caso de que se especifique una arquitectura de destino a la que ya esté portada la biblioteca *Newlib,* como por ejemplo *Linux, RDOS, Novell NetWare*, etc., el código de las funciones *stub* se implementará mediante las llamadas correspondientes al sistema operativo. En el caso contrario, las funciones *stub* por defecto tendrán una implementación mínima que simplemente retornará un valor de error adecuado para indicar al resto de funciones de la biblioteca que los servicios que deberían proporcionar no están disponibles.

Aunque el código por defecto de las funciones *stub* suele ser el adecuado para la mayoría de aplicaciones, hay veces en las que las funciones *stub* generadas por defecto no se ajustan a nuestros propósitos, bien porque no sean compatibles del todo con nuestro sistema empotrado, o bien porque la arquitectura de nuestro sistema no esté soportada. Éste es precisamente el caso de nuestra plataforma de prácticas, la placa *Econotag*. Como la placa tiene un microprocesador ARM7TDMI, cuando se construye la biblioteca *Newlib* se debe especificar la arquitectura *arm-econotag-eabi*, tal y como hicimos en el seminario 2. El problema es que cuando se especifica esta arquitectura a la hora de construir la biblioteca, se genera un código por defecto para las funciones *stub* que realiza llamadas al programa monitor *Angel*, el monitor de depuración que solía venir instalado en las placas de evaluación con procesadores ARM. Si nuestra placa llevara dicho monitor preinstalado, ésta sería la implementación más adecuada para las funciones *stub*, ya que el programa monitor proporcionaría la mayoría de los servicios necesarios. Sin embargo, en nuestro caso la *Econotag* no dispone de ningún programa monitor que pueda satisfacer las llamadas que realizan las funciones *stub* por defecto, por lo que es más conveniente añadir la opción `--disable-newlib-supplied-syscalls` en la construcción de *Newlib* para que no se genere el fichero `syscalls.c` con las funciones *stub* por defecto, tal y como hicimos en el seminario 2.

Con la biblioteca *Newlib* construida de esta forma, si usamos alguna función de *libC* que necesite acceder a los dispositivos de la placa, como por ejemplo `printf`, `abort`, `malloc`, etc., tendremos que implementar en nuestro BSP las funciones *stub* de las llamadas al sistema que proporcionen los servicios necesarios para estas funciones puedan ejecutarse.

La implementación de las funciones *stub* depende de si se ha construido la biblioteca *Newlib* con o sin soporte para llamadas reentrantes. Si no se soportan las llamadas reentrantes (opción por defecto), el nombre de las funciones *stub* será el de la llamada al sistema que implementen anteponiéndole un guión bajo (`_open`, `_kill`, `_fork`, etc.). Por el contrario, si se ha construido la biblioteca para soportar llamadas reentrantes, además de anteponer el guión bajo, también se debe añadir el sufijo `_r` al nombre (`_open_r`, `_kill_r`, `_fork_r`, etc.).

Drivers L2: integración de nuestro BSP con Newlib

La capa L2 de los *drivers* de nuestro BSP estará formada por todas aquellas funciones y estructuras de datos que debemos incluir en nuestro BSP para poder integrarlo adecuadamente con las funciones de E/S de *Newlib*. Estas funciones y estructuras de datos se pueden agrupar en dos partes claramente diferenciadas: el soporte para la E/S estándar, que nos permitirá acceder a los

dispositivos mediante descriptores de fichero y que definirá los ficheros estándar *stdin, stdout* y *stderr*, y la implementación de las llamadas a sistema (`open`, `close`, `read`, `write`, etc.), que se encargarán de llamar a las funciones de los *drivers* L1 del BSP correspondientes en función del descriptor de fichero abierto que se use en cada momento.

Soporte de E/S estándar

La biblioteca *libC* trata a los dispositivos de E/S como si fueran ficheros. Esto implica que para poder realizar una operación de E/S sobre un dispositivo es necesario abrirlo, obtener un descriptor de fichero, y usar dicho descriptor para escribir, leer y volver a cerrarlo cuando hayamos terminado. Todas estas funciones (`open`, `read`, `write`, `close`, …) están definidas en `unistd.h` y se puede consultar su API en las páginas de manual de *Linux/Unix* (página 3).

Existe una excepción para este procedimiento, y es que, en todo sistema, *libC* asume que hay tres ficheros estándar, *stdin, stdout* y *stderr*, que se usan para la entrada estándar, la salida estándar y la comunicación de errores por defecto, y que permanecen siempre abiertos. De hecho, hay funciones como `printf` y `scanf` que asumen que las operaciones de E/S se realizarán sobre estos ficheros estándar, por lo que no es necesario indicarles ningún descriptor de fichero. Los descriptores de fichero que se usan para acceder a *stdin, stdout* y *stderr*, tienen los valores fijos de 0, 1, y 2, respectivamente y están definidos en `unistd.h` como `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`.

Por tanto, si queremos que nuestras aplicaciones puedan hacer uso de las funciones de E/S estándar que proporciona *libC*, tendremos que incorporar a nuestro BSP soporte para poder gestionar los dispositivos como ficheros. Esto implica asignarle a cada dispositivo un nombre (ya que `open` abre los dispositivos a partir de un nombre expresado como una cadena de caracteres), e indicar qué funciones de nuestro BSP se usarán para abrir, cerrar, leer y escribir en dicho dispositivo. También tendremos que gestionar en nuestro BSP una lista de descriptores de ficheros abiertos, dado que *libC* accede a los ficheros mediante descriptores. Además, tendremos que indicar en dicha lista a qué dispositivo están asignadas por defecto sus tres primeras entradas, los descriptores 0, 1 y 2 (`STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`), para que funcionen adecuadamente las funciones de E/S estándar de *libC*.

La lista de dispositivos del sistema

Para facilitar la gestión de los dispositivos de la plataforma, lo más útil es tener en el BSP una lista de dispositivos en la que se almacene, para cada dispositivo, como mínimo, la siguiente información:

- *Nombre del dispositivo*: Será el nombre que se usará para abrir dicho dispositivo mediante la llamada a sistema `open`.
- *Identificador:* En el caso de que haya varios dispositivos del mismo tipo (por ejemplo "`/dev/uart1`" y "`/dev/uart2`"), es necesario indicar para cada uno de ellos el identificador que usan las funciones del *driver* L1 de BSP para diferenciarlos (`uart_1` y `uart_2`, 0 y 1 respectivamente). Este identificador se usará como argumento en la implementación específica de las llamadas a sistema para cada tipo de dispositivo para saber sobre qué dispositivo se debe actuar.
- *Implementación específica de las llamadas a sistema para el dispositivo:* Punteros a las funciones del BSP que implementarán las llamadas a sistema (`open`, `close`, `read`, `write`,

`lseek`, `fstat` e `isatty`) para dicho dispositivo en concreto. Su interfaz (parámetros y tipo de retorno) debe coincidir con la interfaz de la llamada a sistema que implementen, salvo que se cambiará el descriptor de fichero por el identificador de dispositivo en el BSP.

En nuestro BSP se ha definido la estructura `bsp_dev_t` en `hal/dev.h` para gestionar dicha información. La lista de dispositivos, `bsp_dev_list`, se ha definido como un *array* de elementos de tipo `bsp_dev_t` en `hal/dev.c`. Esta implementación es más sencilla y más segura que la implementación mediante una lista enlazada, ya que no usa punteros ni memoria dinámica.

Registro de dispositivos en el BSP

También será necesario un mecanismo sencillo que permita registrar dispositivos en el BSP (esto es, añadir dispositivos a la lista de dispositivos). Para ello, se ha definido la función `bsp_register_dev`, que básicamente rellena la siguiente entrada libre de la lista de dispositivos con la información necesaria para que se pueda gestionar el dispositivo mediante funciones estándar de *libC*.

Teniendo que cuenta que el BSP llama a la función de inicialización de cada uno de los dispositivos antes de llamar a la función `main` de la aplicación, lo más sensato es incluir una llamada a la función `bsp_register_dev` al final del código de inicialización para cada dispositivo del sistema. De esta forma, el registro de los dispositivos se hará automáticamente por el BSP antes de la ejecución del código de la aplicación.

Por último, a la hora de registrar un dispositivo hay que tener en cuenta que se deben facilitar punteros a las funciones del BSP que implementen cada una de las llamadas a sistema para el dispositivo, y que dichas funciones deben tener la misma interfaz que la llamada a sistema que implementen (tipo de datos para el valor de retorno, número de argumentos y tipo de datos para cada uno de los argumentos), salvo el descriptor de fichero, que se cambiará por el identificador de dispositivo en el BSP. En el caso de que el BSP no tenga definida ninguna función específica para implementar alguna llamada a sistema para el dispositivo, se indicará almacenando el valor `NULL` en la entrada correspondiente. En estos casos, cuando no exista una implementación para alguna llamada a sistema para un dispositivo, el BSP ejecutará una implementación mínima por defecto que retornará un valor de error adecuado para asegurar que el comportamiento de la biblioteca *libC* sea coherente.

Búsqueda de un dispositivo por su nombre

Dado que la llamada a sistema `open` debe abrir un dispositivo a partir de su nombre expresado como una cadena de caracteres, es deseable que nuestro BSP implemente una función que devuelva un puntero a la entrada de la lista de dispositivos que contiene dicho dispositivo, en el caso de que el dispositivo indicado esté registrado en el sistema, o bien `NULL`, si el BSP no puede gestionar dicho dispositivo. Esta función está implementada en nuestro BSP mediante la función `find_dev`.

La lista de descriptores de ficheros abiertos

La E/S de bajo nivel de *libC* está basada en el uso de descriptores de fichero. De hecho, si consultamos el API de las llamadas a sistema (`open`, `close`, `read`, `write`, etc.), todas están basadas en el uso de descriptores de fichero. La función `open` retorna un descriptor de fichero una vez que abre con éxito el fichero sobre el dispositivo y el resto de llamadas a sistema usan dicho descriptor para identificar el fichero sobre el que deben actuar.

Puesto que es posible abrir más de un fichero sobre el mismo dispositivo, será necesario mantener en el BSP una lista que almacene información sobre todos los ficheros que hay abiertos en cada momento, indicando sobre qué dispositivo se ha abierto cada uno, y qué *flags* se han usado para abrirlo, de forma que se pueda acceder a esta información cuando se necesite leer o escribir información en el fichero.

En nuestro BSP se ha definido la estructura `bsp_fd_t` en `hal/dev.h` para gestionar dicha información. Al igual que en el caso de los dispositivos, la lista de descriptores de ficheros abiertos, `bsp_fd_list`, se ha definido como un *array* de elementos de tipo `bsp_fd_t` en `hal/dev.c`.

Puesto que un descriptor de fichero no es más que un valor entero que indica de forma única un fichero abierto sobre un dispositivo, en nuestro BSP se usará el índice de la entrada que almacene los datos del fichero en la lista de ficheros abiertos del BSP como descriptor de dicho fichero.

Obtención del dispositivo y los *flags* de apertura de un fichero

Dado un descriptor de fichero, para obtener el dispositivo sobre el que está abierto, así como sus *flags* de apertura, simplemente hay que consultar la lista de descriptores de fichero del BSP, usando el propio descriptor como índice para encontrar la entrada adecuada de la lista. Para facilitar esta tarea, nuestro BSP incorpora las funciones `get_dev` y `get_flags`.

Asignación y liberación de descriptores de fichero

Cuando se abre un nuevo fichero, la llamada a sistema `open` debe retornar un descriptor de fichero válido a la aplicación. Para ayudar a la implementación de esta llamada a sistema, nuestro BSP cuenta con la función `get_fd`, que encuentra la primera entrada disponible de la lista de descriptores del BSP, almacena en ella los datos relativos a la apertura del fichero, y retorna el índice de dicha entrada en la lista para que se use como descriptor del fichero.

Puesto que el tamaño de la lista de descriptores de fichero del BSP tiene un tamaño limitado, y que a lo largo de la ejecución de la aplicación se espera que se abran y se cierren diferentes ficheros, es necesario implementar un mecanismo sencillo y dinámico de asignación de descriptores de fichero. Dado que inicialmente estarán todas las entradas de la lista de descriptores inicializadas a cero, la función `get_fd` de nuestro BSP, hará una búsqueda secuencial en la lista de descriptores hasta que encuentre una entrada en la que el campo `dev` sea `NULL`. Una vez encontrada dicha entrada, almacenará en el campo `dev` el puntero al dispositivo sobre el que se va a abrir el fichero, en el campo `flags` sus *flags* de apertura, y retornará el índice de dicha entrada.

Por otro lado, cuando se cierra un fichero se debe liberar la entrada correspondiente en la lista de descriptores. Para ello, nuestro BSP cuenta con la función `release_fd`, que dado un descriptor de fichero, lo usa como índice para acceder a la lista de ficheros abiertos, y fija sus campos `dev` y `flags` a `NULL` y 0 respectivamente, liberando dicha entrada para que pueda ser asignada a otro fichero en un futuro.

Ficheros de E/S estándar por defecto y redirección de la E/S estándar

Como la biblioteca *libC* supone que siempre van a existir los ficheros estándar *stdin*, *stdout* y *stderr*, es necesario que el BSP mantenga abiertos tres ficheros, con descriptores 0, 1 y 2, sobre algún dispositivo por defecto, de forma que en el caso de que no se llegue a registrar ningún dispositivo en el BSP, siempre puedan ejecutarse todas las funciones de la biblioteca sin causar situaciones de error.

Para satisfacer esta restricción, el BSP cuenta con un dispositivo por defecto. El dispositivo `dev/null`[3], para el que las llamadas a sistema usan la implementación mínima por defecto del BSP, es decir, que es un fichero de caracteres, que no se puede ni abrir ni cerrar, que no se puede leer de él, aunque acepta cualquier dato que se le envíe, y que no se puede alterar el desplazamiento dentro del fichero.

Los ficheros de E/S estándar, *stdin*, *stdout* y *stderr*, estarán abiertos sobre `/dev/null` por defecto, de forma que las llamadas a las funciones de E/S de *libC* no fallen aunque no haya dispositivos registrados en el BSP. Para ello, simplemente hay que asignar las tres primeras entradas de la lista de descriptores de fichero del BSP al dispositivo `/dev/null`, tal y como se muestra en `hal/dev.c`.

Sin embargo, una vez que se han registrado dispositivos en el BSP, lo más sensato es redireccionar la E/S estándar para asignar *stdin*, *stdout* y *stderr* a los dispositivos que nos interese (puertos serie, pantallas, teclados, etc.). Para ello, simplemente hay que cambiar las tres primeras entradas de la lista de ficheros abiertos para que almacenen los dispositivos y los *flags* de apertura adecuados. Nuestro BSP proporciona la función `redirect_fd` para facilitar esta tarea.

Llegados a este punto, es conveniente echar un vistazo a la función de inicialización del BSP, `bsp_init` en `hal/bsp_init.c`, donde se puede comprobar cómo se inicializan los dispositivos y se redirecciona la E/S estándar del BSP usando las funciones de gestión de dispositivos y ficheros que se han descrito en esta práctica.

Implementación de las llamadas a sistema de *Newlib*

Una vez que nuestro BSP dispone de toda la infraestructura necesaria para gestionar ficheros, ha llegado el momento de completar el fichero `hal/syscalls.c` de nuestro BSP con la implementación de las llamadas a sistema de *Newlib* necesarias para terminar de dar soporte a la E/S estándar. Para ello hay que recordar que, dado que hemos construido *Newlib* sin soporte para llamadas reentrantes, las llamadas a sistema de *Newlib* están implementadas como llamadas a funciones *stub* que debemos implementar en nuestro BSP, y que el nombre de dichas funciones debe ser el de la llamada a sistema que implementen anteponiéndole un guión bajo.

En general, la implementación de una función *stub* para una llamada a sistema sigue siempre el mismo esquema: identificar el dispositivo sobre el que se debe actuar, y si el dispositivo tiene definida una implementación específica para la llamada a sistema, llamar a dicha implementación. En caso contrario, se ejecutará la implementación mínima por defecto de la llamada a sistema para garantizar un correcto funcionamiento de la biblioteca *libC*. La implementación mínima de las llamadas a sistema se puede consultar en la documentación de *Newlib*[4].

Implementación de _open

La llamada a sistema `open` recibe un nombre de dispositivo junto con unos *flags* y un modo de apertura. En caso de éxito, retornará el descriptor del fichero abierto, y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Lo primero que debe hacer su implementación es consultar en la lista de dispositivos si existe un dispositivo con el nombre especificado (mediante la función `find_dev` del BSP). En el caso de que el dispositivo exista, habrá que consultar en la entrada correspondiente de la lista de dispositivos si tiene definida una implementación específica de la función `open`, y si es así, se llamará a dicha

3 Dispositivo nulo. http://es.wikipedia.org/wiki//dev/null
4 Sección 12 de http://sourceware.org/newlib/libc.html.

función, que supuestamente realizará las tareas necesarias en el dispositivo para configurarlo adecuadamente. Si la implementación de `open` para el dispositivo no falla (es decir, si retorna un valor no negativo), es señal de que la apertura se ha realizado correctamente, por lo que se debe llamar a la función `get_fd` del BSP para obtener un descriptor de fichero válido para el fichero y se retornará dicho descriptor.

Si falla alguna de estas condiciones, bien porque no existe un dispositivo con dicho nombre en el sistema, o porque no tenga una implementación específica para la función `open`, o porque la implementación de su función `open` falle, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `open`, que básicamente retornar -1 (retorno con error) y fijar el código correspondiente en la variable global `errno`. Si el dispositivo no existe se fijará `errno` al valor `ENODEV`, y si el dispositivo no tiene definida la función `open` se fijará errno al valor `ENOTSUP` (operación no soportada).

Implementación de _close

La llamada a sistema `close` recibe el descriptor del fichero que debe ser cerrado. En caso de éxito, retornará 0, y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Lo primero que debe hacer su implementación es liberar dicho descriptor para que pueda ser usado en el futuro por otros ficheros (mediante la función `release_fd` del BSP). Después, se debe consultar en la lista de descriptores de fichero sobre qué dispositivo está abierto dicho fichero (mediante la función `get_dev` del BSP), y si el dispositivo existe en la lista de descriptores, y además dispone de una implementación específica para la llamada a sistema `close`, se llamará a dicha implementación y se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `close`, que básicamente consiste en fijar la variable global de código de error `errno` al valor `EBADF` (el descriptor de fichero no se corresponde con un fichero abierto válido), y retornar -1 (retorno con error).

Implementación de _read

La llamada a sistema `read` recibe el descriptor del fichero sobre el que se desea leer, un puntero a un búfer para almacenar los datos leídos, y el número de bytes que se desea leer. En caso de éxito, retornará el número de bytes que se han podido leer del fichero[5], y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Si el descriptor de fichero es válido y el dispositivo sobre el que está abierto tiene una implementación específica para la llamada a sistema `read`, se llamará a dicha implementación y se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `read`, que consiste en retornar un 0, es decir, que no se ha podido leer ningún byte del fichero.

Implementación de _write

La llamada a sistema `write` recibe el descriptor del fichero sobre el que se desea escribir, un puntero a un búfer con datos, y el número de bytes que se desea escribir en el fichero. En caso de

5 Puede que aunque se hayan leído algunos bytes, no se hayan podido leer todos los bytes que se indicaba en los parámetros de la llamada.

éxito, retornará el número de bytes que se han podido escribir del fichero[6], y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Si el descriptor de fichero es válido y el dispositivo sobre el que está abierto tiene una implementación específica para la llamada a sistema `write`, se llamará a dicha implementación y se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `write`, que consiste en retornar el total de datos que se quería escribir en el fichero, aunque no se haya escrito nada.

Implementación de _lseek

La llamada a sistema `lseek` recibe el descriptor del fichero sobre el que se desea realizar el desplazamiento, un desplazamiento, y una base (desde el inicio, desde el fin o desde la posición actual). En caso de éxito, retornará el número de bytes que tiene el nuevo desplazamiento desde el inicio del fichero, y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Si el descriptor de fichero es válido y el dispositivo sobre el que está abierto tiene una implementación específica para la llamada a sistema `lseek`, se llamará a dicha implementación y se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `lseek`, que consiste en retornar un 0, es decir, se asume que el desplazamiento estará siempre al comienzo del fichero y que no se puede cambiar.

Implementación de _fstat

La llamada a sistema `fstat` recibe un descriptor de fichero y un puntero a una estructura de tipo `stat`. En caso de éxito, retornará un 0 y rellenará la estructura con información relativa al fichero, y en caso de fallo, retornará -1 y fijará en la variable global `errno` el código del error que se ha producido.

Si el descriptor de fichero es válido y el dispositivo sobre el que está abierto tiene una implementación específica para la llamada a sistema `fstat`, se llamará a dicha implementación y se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `fstat`, que consiste rellenar el campo `st_mode` de la estructura al valor `S_IFCHR`, para indicar que el fichero está abierto sobre un dispositivo de caracteres, y retornar un 0.

Implementación de _isatty

La llamada a sistema `isatty` recibe un descriptor de fichero y retorna 1 si el fichero es un terminal o 0 en otro caso.

Si el descriptor de fichero es válido y el dispositivo sobre el que está abierto tiene una implementación específica para la llamada a sistema `isatty`, se retornará el valor de retorno de esta implementación específica. En cualquier otro caso, se debe ejecutar la implementación mínima por defecto de la llamada a sistema `isatty`, que consiste en retornar un 1.

6 Al igual que en la lectura, en la escritura puede que no se hayan podido escribir todos los datos en el fichero.

Soporte de memoria dinámica

Aunque esta práctica está orientada a proporcionar un *driver* L2 para las UART de la *Econotag*, y *a priori* nos pueda parecer que la E/S y la gestión de memoria dinámica son dos cosas totalmente independientes, lo cierto es que las funciones de la familia de `printf` de *Newlib* hacen uso de `malloc` para reservar un búfer que les ayude a procesar la cadena de formato.

Por tanto, si queremos hacer uso de estas funciones, es necesario añadir el soporte adecuado a nuestro BSP para gestionar adecuadamente la zona de memoria de nuestra plataforma que se ha reservado como *heap*. Para ello necesitaremos hacer uso de los símbolos globales `heap_start` y `heap_end`, definidos en nuestro *linker script*, e implementar la llamada a sistema `sbrk`, que se encarga de incrementar el tamaño del área reservada para datos dentro del *heap*. La implementación de la función *stub* que da soporte a la llamada a sistema `sbrk` está disponible en `hal/syscalls.c`.

Ejercicios

El objetivo final de esta serie de ejercicios es conseguir integrar totalmente nuestro BSP con la biblioteca *libC* de *Newlib*, de forma que podamos gestionar las UART de nuestra plataforma mediante las funciones estándar del *libC*.

Comprender el API de gestión de dispositivos y descriptores de fichero del BSP

Debido al escaso tiempo de que disponemos para hacer prácticas, la gestión de dispositivos y descriptores de fichero se proporciona completamente implementada en el fichero `hal/dev.c`, por lo que en esta práctica nos centraremos en el desarrollo de las llamadas a sistema. Sin embargo, y dado que las llamadas a sistema harán uso del API de gestión de dispositivos y descriptores de fichero del BSP, se recomienda que se revise y se entienda adecuadamente dicha API a fin de poder completar los objetivos de la práctica correctamente.

Implementar las llamadas a sistema necesarias para proporcionar una E/S estándar

Implementar las funciones *stub* `_open`, `_close`, `_read`, `_write`, `_lseek`, `_fstat` e `_isatty` para completar la integración de nuestro BSP con la biblioteca *Newlib*.

Registro de las UART en la lista de dispositivos del BSP

Completar la implementación de la función `uart_init` del *driver* de las UART para que las registre en la lista de dispositivos del BSP.

Para ello, y dado que en las dos últimas prácticas hemos ido añadiendo lineas a la función `uart_init` para dar soporte a los driver L0 y L1, sólo nos queda insertar, justo antes de retornar, una llamada a la función `bsp_register_dev` indicando el nombre del dispositivo (el que se le ha pasado a `uart_init` en el parámetro `name`[7]), su identificador (el que se le ha pasado a `uart_init` en el parámetro `uart`), y las funciones del BSP que implementarán las llamadas a sistema para las UART. Se usará la implementación mínima por defecto para todas las llamadas a sistema excepto para `read` y `write`, que se implementarán mediante las llamadas `uart_receive` y `uart_send`. Para ello, al diseñar las funciones `uart_receive` y `uart_send` en la práctica anterior se ha tenido especial cuidado para que tanto la lista de los argumentos como los valores de retorno coincidan con los de las llamadas a sistema que van a implementar.

7 Los nombres de los dispositivos, así como todos sus parámetros de configuración están definidos en `bsp/system.h`.

Prueba del *driver* L2

Reescribir la aplicación de la práctica anterior usando solamente funciones estándar de *libC* (`getchar`, `printf`, etc.) para acceder a la UART de la placa.