Práctica 9

Desarrollo de un driver L1 para las UART

Introducción

Una vez que tenemos la implementación del *driver* L0 para las UART, ha llegado el momento de sacarle partido a la gestión de excepciones e interrupciones que tenemos en el BSP para implementar el *driver* L1, lo que nos permitirá realizar una E/S de altas prestaciones de forma concurrente con la ejecución de nuestra aplicación. Para ello, nuestro *driver* implementará funciones de E/S no bloqueantes que harán uso de búferes circulares e interrupciones, tal y como se ha mostrado en las lecciones de teoría.

Objetivos

- Saber hacer uso de búferes circulares para implementar funciones de E/S no bloqueantes
- Saber diseñar una ISR que gestione las transferencias entre los búferes circulares y el dispositivo de E/S de forma concurrente con la ejecución de la aplicación
- Saber usar funciones *callback* para gestionar la E/S desde la aplicación

Drivers L1

Toda implementación de un *driver* L1 se apoya en el uso intensivo dos búferes circulares, uno para los envíos y otro para las recepciones, y proporciona a la aplicación dos funciones, `send` y `receive`, que se encargan de escribir y leer respectivamente de dichos búferes. Una vez que estas funciones operan sobre los búferes retornan a la aplicación, que continúa con su ejecución, confiando en que las operaciones transferencia se realizarán en segundo plano de forma concurrente. Sin embargo, y dado que normalmente la tasa de transferencia de los dispositivos es más lenta que la copia de los datos en los búferes, lo que ocurre realmente es que los datos permanecen un tiempo en los búferes desde que la aplicación llama a `send` hasta que realmente son enviados, o bien desde que se reciben por el dispositivo hasta que la aplicación los recibe.

Las transferencias entre los búferes circulares del *driver* y el dispositivo se realizan mediante la ISR del dispositivo, cuya ejecución es solicitada por el dispositivo cada vez que está preparado para enviar o bien cuando acaba de recibir datos. Este proceso se realiza mediante la petición de interrupción por parte del dispositivo, que una vez que pasa por el controlador de interrupciones y llega a la CPU, causa la interrupción momentánea de la ejecución de la aplicación para servir la interrupción, lo que implica ejecutar la ISR del dispositivo y retornar a la ejecución normal de la aplicación. Para que el proceso funcione correctamente y la ejecución de la aplicación no se vea penalizada, las ISR deben ser muy ligeras.

Por último, otra característica importante de un *driver* L1 es proporcionar a la aplicación la posibilidad de poder definir funciones *callback* para gestionar adecuadamente las operaciones de E/S. La función *callback* de recepción será llamada por la ISR cuando el dispositivo haya recibido datos, y una vez que la ISR los haya copiado al búfer circular de recepción, por lo que normalmente se suele utilizar para que la aplicación recoja los datos recibidos (mediante la función `receive`). La

ISR también llamará a la función *callback* de envío cuando se hayan enviado los datos pendientes de envío del búfer circular, para indicarle a la aplicación que puede enviar más datos (mediante `send`). De esta forma se facilita el uso del dispositivo por el desarrollador de aplicaciones, ya que el *driver* está basado en una ISR genérica que avisa a la aplicación cuando se ha terminado de enviar los datos o cuando han llegado datos nuevos. El desarrollador ya no tiene que lidiar con las complicaciones de escribir una ISR. Simplemente debe escribir dos funciones *callback* que serán llamadas automáticamente cada vez que el dispositivo necesite más datos (*callback* de envío) o cuando el dispositivo reciba datos nuevos (*callback* de recepción).

Además del soporte de interrupciones, otra característica importante de los *drivers* L1 es el chequeo de errores. Las funciones del *driver*, antes de implementar la lógica necesaria para llevar a cabo su función, chequearán si los parámetros son correctos, y en el caso de que no lo sean, retornarán con un código de error, indicando a la aplicación la causa del problema.

Gestión de los búferes circulares

Se asume que la capacidad para diseñar y desarrollar búferes circulares se ha adquirido con anterioridad en otras asignaturas de la titulación, por lo que no se cubrirá en esta práctica. Sin embargo, y aunque cada alumno puede implementarlos como desee, para facilitar el desarrollo de la práctica, en `util/circular_buffer.c` se proporciona una implementación de un búfer circular que se usa dentro del código de nuestro *driver* de la UART. Concretamente, y dado que nuestra plataforma dispone de dos UART, al principio del fichero `drivers/uart.c` se definen las siguientes estructuras:

```
static volatile circular_buffer_t uart_circular_rx_buffers[uart_max];
static volatile circular_buffer_t uart_circular_tx_buffers[uart_max];
```

donde `uart_circular_rx_buffers[i]` será una estructura de tipo `circular_buffer_t` que contendrá el búfer circular de recepción de la uart i-ésima. Análogamente, `uart_circular_tx_buffers[i]` será su búfer circular de transmisión.

Si consultamos el fichero `include/circular_buffer.h` podremos consultar el API de los búferes circulares, consistente en una función de inicialización, funciones para leer y escribir en el búfer, y funciones para chequear si el búfer está lleno o vacío. Es importante entender bien el funcionamiento de este API, ya que se usará para soportar la gestión de los búferes circulares de nuestro *driver*.

Envío de datos

Como todas las funciones del API pública de un *driver* L1, lo primero que debe hacer esta función es comprobar que los parámetros de entrada son válidos. En nuestro caso, habrá que retornar con error (–1) si:

- El parámetro `uart >= uart_max`, ya que sólo tenemos dos UART. En este caso se fijará la variable `errno` a `ENODEV`, indicando que no existe tal dispositivo.
- El puntero al búfer es `NULL` o `count` es negativo, ya que en ambos casos no se podría realizar la copia de los datos. Es este caso se fijará la variable `errno` a `EFAULT`, indicando que los datos relativos al búfer no son válidos.

Los valores anteriores para la variable global `errno` están definidos en el fichero `errno.h` de la biblioteca estándar de *C*.

Si los valores de los parámetros son válidos, podemos proceder a copiar todos los bytes que podamos desde el búfer de la aplicación al búfer circular del *driver*. El proceso de copia se realizará mientras que el búfer circular de envío no esté ya lleno y queden datos por copiar en el búfer de la aplicación.

Por otro lado, y teniendo en cuenta que en cualquier momento de la ejecución podría ocurrir una interrupción de la misma UART, debemos realizar la copia en una región crítica, pero de forma que se afecte mínimamente al funcionamiento del sistema. Para ello, deshabilitaremos la petición de interrupciones del transmisor de la UART mientras realizamos la copia de datos al búfer circular de envío y volveremos a habilitarla una vez que la copia haya terminado. De esta forma evitaremos que la ISR de la UART interfiera en la copia de los datos al búfer circular, aunque sí que podría interrumpir dicha copia para atender una recepción de datos, ya que la recepción involucra la modificación de otro búfer circular diferente. La habilitación/deshabilitación de la petición de interrupciones del transmisor de la UART se gestiona con el bit `mTxR` del registro `UCON`.

Por último, y teniendo en cuenta que puede que no hayamos podido copiar en el búfer circular todos los datos que nos ha pedido la aplicación, la función `send` debe retornar el número de bytes que se han podido copiar al búfer circular del *driver*.

Recepción de datos

Al igual que en el caso del envío, se deben chequear los valores de los parámetros de la función. Los tests serán los mismos que en el caso del envío. Se debe comprobar que el identificador de la UART es válido, que el puntero al búfer de datos no es `NULL` y que `count` no es negativo.

Una vez pasados los tests, podemos proceder a copiar todos los bytes que podamos desde el búfer circular del *driver* al búfer de la aplicación. El proceso de copia se realizará mientras que el búfer circular de recepción no esté vacío y queden datos por copiar al búfer de la aplicación. Al igual que en el envío de datos, debemos evitar que la ISR de la UART interfiera en el proceso de copia, por lo que deberemos establecer otra región crítica para el acceso al búfer circular de recepción. Para ello, y dado que la ISR intentará acceder a dicho búfer sólo en el caso de que se reciban datos en el dispositivo, debemos deshabilitar la petición de interrupciones del receptor de la UART mientras realizamos la copia de datos desde el búfer circular al búfer de la aplicación y volver a habilitarla una vez que la copia haya terminado. De esta forma la penalización en el sistema es mínima, ya que, cualquier otro dispositivo, o incluso la ISR de la UART podrán interrumpir la copia de forma segura, ya que estarán accediendo a otras estructuras de datos diferentes. La habilitación/deshabilitación de la petición de interrupciones del receptor de la UART se gestiona con el bit `mRxR` del registro `UCON`.

Al igual que en el envío, puede que no se hayan podido copiar al búfer de la aplicación todos los bytes solicitados, por lo que la función `receive` debe retornar el número de bytes que se han podido copiar desde el búfer circular del *driver*.

Definición de las funciones callback

Como se ha comentado más arriba, las funciones *callback* son funciones de la aplicación que serán llamadas por la ISR del *driver* cuando se reciban datos en el dispositivo o bien cuando el dispositivo esté preparado para poder transmitir más datos.

Si observamos el fichero `drivers/uart.c`, podemos ver que para cada UART se define una estructura de tipo `uart_callback_t` que contiene dos punteros a función, uno para la función *callback* de transmisión y otro para la de recepción. Como disponemos de dos UARTS, se ha definido el *array* `uart_callbacks[]`, en el que cada elemento es una estructura `uart_callback_t`, de forma que se facilita el acceso a la función *callback* que necesitemos de cada una de las UART.

Inicialmente, los punteros a las funciones *callback* de cada UART tienen el valor `NULL`, lo que indica que no se han definido dichas funciones. Para poder asignar funciones *callback* a una UART nuestro *driver* tiene definidas las funciones `uart_set_receive_callback` y `uart_set_send_callback`, que reciben un identificador de una UART y un puntero a función. Dichas funciones deben comprobar que el identificador de la UART es válido, y si es así, asignar el puntero de la función *callback* a la UART correspondiente.

Servicio de interrupciones

Cada dispositivo del sistema debe tener su propia función ISR, ya que cuando ocurre una interrupción, el manejador de excepción correspondiente consulta al controlador de interrupciones, obtiene el identificador del dispositivo que ha pedido la interrupción, y llama a su ISR para que se sirva la petición del dispositivo. Para poder implementar este mecanismo correctamente las ISR deben ser funciones sin argumentos que no retornen nada, y deben estar codificadas para poder acceder a los registros de control del dispositivo al que sirvan, así como a las estructuras de datos del *driver* que gestiona dicho dispositivo.

Como en nuestro caso disponemos de dos UART, en el fichero `driver/uart.c` se han definido dos ISR, `uart_1_isr` y `uart_2_isr`. Para facilitar la gestión de estas ISR desde nuestro *driver*, se ha creado el *array* `uart_irq_handlers[]`, que contiene los punteros a las ISR de las UART del sistema, de forma que podamos acceder a ellas mediante el identificador de cada UART.

En principio, la implementación de la ISR de cada UART podría ser diferente, ya que cada UART dispone de su propia ISR. Sin embargo, como en este caso queremos que nuestro *driver* gestione las dos UART de la misma forma, las ISR de las UART se han implementado como llamadas a una ISR genérica:

```
static void uart_1_isr (void) { uart_isr (uart_1); }
static void uart_2_isr (void) { uart_isr (uart_2); }
```

por lo que a continuación nos dedicaremos a describir qué debe hacer dicha ISR para servir adecuadamente a las UART del sistema.

Si miramos el manual de referencia del MC13224V de *Freescale*, podemos observar que, si la generación de interrupciones está habilitada, las UART generarán una interrupción en alguno de estos casos:

- Si se ha producido algún error en el envío/recepción de datos. Los errores detectados se marcan activando los bits `RuE`, `RoE`, `ToE`, `FE`, `PE` o `SE` del registro `USTATUS`.
- Si se han recibido nuevos datos en el dispositivo. Concretamente, si el nivel de datos recibidos en la cola FIFO de recepción del dispositivo excede el límite especificado en el campo `RxLevel` del registro `URxCON`. La recepción de nuevos datos se indica activando el bit

RxRdy del registro USTATUS.

- Si el dispositivo necesita más datos para ser enviados. Concretamente, si el espacio vacío de la cola FIFO de envío del dispositivo excede el límite especificado en el campo TxLevel del registro UTxCON. El agotamiento de los datos se indica activando el bit TxRdy del registro USTATUS.

Estas causas no son excluyentes, por lo que la ISR deberá comprobar cada una de ellas, y para cada una que esté activa, deberá reconocerla (para que el dispositivo deje de solicitar la interrupción) y darle servicio.

Tratamiento de errores

El reconocimiento de una interrupción causada por un error de transmisión/recepción se realiza simplemente leyendo el registro USTAT, en el que se indica qué tipo de error se ha producido. Una vez que se lee dicho registro, la UART dejará de solicitar la interrupción. Chequeando los bits 5:0 leídos del registro USTAT se puede determinar la causa del error y darle la solución adecuada. Sin embargo, si el *driver* está bien diseñado es muy poco probable que ocurran errores, por lo que en esta práctica omitiremos el tratamiento de los errores en la ISR, aunque en cualquier caso será necesario leer el registro USTAT al principio de la ISR para que en el caso poco probable de que se haya producido un error de transmisión/recepción, se reconozca y el dispositivo deje de solicitar una interrupción por esta causa.

Recepción de datos

Si el nivel de datos recibidos en la cola FIFO de recepción del dispositivo excede el límite especificado en el campo RxLevel del registro URxCON, la UART activará el bit RxRdy del registro USTATUS y provocará una interrupción. Para reconocer dicha interrupción simplemente hay que leer datos de la cola FIFO.

Teniendo esto en cuenta, la ISR deberá testar el bit RxRdy del registro USTATUS, y en el caso de que esté activo, copiar bytes de la cola FIFO del dispositivo al búfer circular de recepción del *driver* mientras que haya espacio en el búfer circular y queden datos en la cola FIFO. Para ello hay que tener presente que la lectura del campo Rx_data del registro UDATA saca un dato de la cola FIFO, y que el número de bytes almacenados en la cola FIFO de recepción se puede consultar leyendo el campo Rx_fifo_addr_diff del registro URxCON.

Una vez que se han copiado todos los bytes que se ha podido al búfer circular, es el momento de llamar a la función *callback* de recepción de la UART (en el caso de que esté definida) para indicarle a la aplicación que el *driver* tiene nuevos datos almacenados en su búfer circular.

Por último, en el caso de que el búfer circular de recepción del *driver* esté lleno, se debe enmascarar la petición de interrupciones del receptor de la UART (bit mRxR del registro UCON) para que no interrumpa la ejecución del programa si llegan más datos, ya que no podrán almacenarse en el búfer circular mientras siga lleno.

Envío de datos

Si el espacio libre en la cola FIFO de envío del dispositivo excede el límite especificado en el campo TxLevel del registro UTxCON, la UART activará el bit TxRdy del registro USTATUS y provocará una interrupción. Para reconocer dicha interrupción simplemente hay que escribir datos en la cola FIFO.

Teniendo esto en cuenta, la ISR deberá testar el bit `TxRdy` del registro `USTATUS`, y en el caso de que esté activo, copiar bytes del búfer circular de transmisión del *driver* a la cola FIFO de envío del dispositivo mientras que haya espacio en la cola FIFO y queden datos en el búfer circular. Para ello hay que tener presente que la escritura en el campo `Tx_data` del registro `UDATA` mete un dato en la cola FIFO, y que el número de bytes vacíos de la cola FIFO se puede consultar leyendo el campo `Tx_fifo_addr_diff` del registro `UTxCON`.

Una vez que se han copiado todos los bytes que se ha podido a la cola FIFO de envío del dispositivo, es el momento de llamar a la función *callback* de envío de la UART (en el caso de que esté definida) para solicitar más datos a la aplicación.

Por último, en el caso de que el búfer circular de envío del *driver* esté vacío, se debe enmascarar la petición de interrupciones del transmisor de la UART (bit `mTxR` del registro `UCON`) para que no interrumpa la ejecución del programa si su cola FIFO se agota, ya que mientras que el búfer circular esté vacío no habrá más datos que transmitir.

Inicialización de las UART

Una vez que hemos añadido todas las funciones que convierten nuestro *driver* en un *driver* L1, es necesario modificar la función de inicialización de las UART para añadirle la gestión de errores y la inicialización de los búferes circulares, el controlador de interrupciones, y la generación de peticiones de interrupción por parte del dispositivo.

En cuanto al chequeo de errores, antes de inicializar el dispositivo la función de inicialización debe comprobar que el identificador de la UART es válido y que el nombre del dispositivo no es `NULL`. En el caso de que alguna de estas condiciones se incumpla se debe retornar con error y se debe fijar en la variable global `errno` el código adecuado. Si el identificador de la UART no es válido se debe fijar la variable `errno` a `ENODEV` y si el nombre del dispositivo es `NULL` se fijará a `EFAULT`[1].

Una vez implementado el chequeo de errores, pasamos a inicializar los búferes y la gestión de interrupciones del *driver*. El código para realizar estas tareas se insertará justo después del código de inicialización L1 que ya teníamos de la práctica anterior.

El primer paso para soportar las funciones L1 consiste en inicializar los búferes circulares de nuestro *driver*. Para ello tendremos que llamar a la función `circular_buffer_init` tanto para el búfer de transmisión (`uart_circular_tx_buffers[uart]`) como para el búfer de recepción (`uart_circular_rx_buffers[uart]`) de nuestra UART.

Después indicaremos a la UART cuándo debe solicitar interrupciones al procesador. Para ello se usan los campos `TxLevel` del registro `UTxCON` y `RxLevel` del registro `URxCON`. El primero fija cuántos bytes vacíos debe tener la cola FIFO de transmisión del dispositivo para interrumpir a la CPU para pedir más datos, y el segundo indica cuántos bytes deben haberse recibido en la cola FIFO de recepción para interrumpir a la CPU y solicitarle que los retire. Teniendo en cuenta que las colas FIFO son de 32 bytes, fijaremos `TxLevel` a 31 y `RxLevel` a 1.

Una vez configurada la generación de interrupciones en la UART, pasamos a configurar la gestión de dichas interrupciones por el controlador de interrupciones del sistema. Tenemos que indicarle al controlador ITC que asigne la UART a la entrada *IRQ* de la CPU (mediante la función `itc_set_priority`). También debemos asignar (mediante `itc_set_handler`) a dicha fuente de interrupción la ISR que se usará para servir al dispositivo (`uart_irq_handlers[uart]`). Por último, debemos habilitar en el controlador ITC el servicio de las interrupciones de la UART

1 Realmente, el nombre del dispositivo no se usará hasta que implementemos el *driver* L2.

(mediante `itc_enable_interrupt`).

Una vez configurada la gestión de interrupciones en el controlador ITC, pasamos a inicializar las funciones *callback* del *driver* (`uart_callbacks[uart]`), que por defecto se inicializarán a `NULL` para indicar que inicialmente no hay funciones *callback* definidas.

Por último, sólo nos queda habilitar la generación de interrupciones en el caso de recepción de datos en la UART (mediante el campo `mRxR` del registro `UCON`).

Adaptación de las funciones bloqueantes del driver L0

En la práctica anterior se desarrollaron dos funciones bloqueantes, `uart_send_byte` y `uart_receive_byte`, que accedían directamente a las colas FIFO del dispositivo para enviar y recibir bytes. Una vez que se ha añadido el soporte de interrupciones al *driver,* es la ISR la que accede a las colas FIFO, mientras que las funciones de enviar y recibir operan sobre los búferes circulares. Para evitar interferencias entre la ISR de las UART, que podría ejecutarse en cualquier momento, y las funciones bloqueantes, es necesario adaptarlas como se describe a continuación.

Envío bloqueante de un byte

Para evitar que la ISR interrumpa a la función `uart_send_byte`, lo primero que debe hacer dicha función es deshabilitar las interrupciones del transmisor de la UART (modificando el bit `mTxR` del registro `UCON`).

Después, en el caso de que haya escrituras pendientes en el búfer circular de envío, se deberán enviar todas a la cola FIFO, ya que estas escrituras se ordenaron con anterioridad a la llamada a `uart_send_byte`.

Una vez que nos aseguramos de que el búfer circular de envío está vacío, pasamos a ejecutar el código de la función original, esto es, esperar a que haya algún hueco en la cola FIFO y cuando así sea, escribir el dato directamente en la cola FIFO.

Por último, debemos dejar el bit `mTxR` del registro `UCON` como estaba antes de deshabilitar las interrupciones del transmisor (obviamente, para ello tendremos que haber almacenado una copia antes de deshabilitar las interrupciones).

Recepción bloqueante de un byte

En la recepción bloqueante también hay que evitar las posibles interferencias con la ISR, por lo que al igual que en el envío, el primer paso de la función `uart_receive_byte` consiste en deshabilitar las interrupciones del receptor de la UART (modificando el bit `mRxR` del registro `UCON`).

Después, si hubiera datos pendientes de ser recibidos en el búfer circular de recepción, extraeremos el byte de ahí. En caso contrario, tendríamos que pasar a ejecutar el código de la función original, esto es, esperar a que haya algún dato en la cola FIFO y cuando así sea, sacarlo de la cola FIFO.

Por último, antes de retornar el byte a la aplicación debemos dejar el bit `mRxR` del registro `UCON` como estaba antes de deshabilitar las interrupciones del transmisor.

Ejercicios

El objetivo final de esta serie de ejercicios es mejorar el *driver* de las UART de nuestro BSP para mejorar sus prestaciones mediante la implementación de llamadas no bloqueantes que harán uso de

búferes circulares e interrupciones.

Envío de datos

Implementar la función `uart_send`, la función de envío no bloqueante de nuestro *driver* L1.

Recepción de datos

Implementar la función `uart_receive`, la función de recepción no bloqueante de nuestro *driver* L1.

Gestión de las funciones *callback*

Implementar las funciones `uart_set_receive_callback` y `uart_set_send_callback`.

Servicio de las interrupciones

Implementar la ISR genérica `uart_isr` de nuestro *driver* L1.

Inicialización de las UART

Modificar la función `uart_init` para que soporte gestión de errores e inicialice los búferes y los mecanismos de gestión de interrupciones del *driver* L1.

Adaptación de las funciones de E/S bloqueantes de las UART

Modificar las funciones `uart_send_byte` y `uart_receive_byte` para que evitar las interferencias de la ISR del *driver* L1.

Prueba del *driver*

Una vez que tenemos el *driver* terminado es el momento de comprobar si funciona. Para ello, y apoyándonos en el código desarrollado en las prácticas anteriores, en este ejercicio se propone la creación de una aplicación que, basándose en el contenido de dos variables globales (por ejemplo `blink_red_led` y `blink_green_led`) haga parpadear los leds de la placa. Inicialmente, dichas variables estarán a `TRUE`, por lo que la aplicación debe parpadear los dos leds ininterrumpidamente.

El estado de las variables globales `blink_red_led` y `blink_green_led` sólo se cambiará mediante una función *callback* de recepción de la UART, que se definirá y se instalará por la aplicación para implementar una sencilla interfaz al usuario vía UART que le permitirá modificar el estado de los leds, de forma que pulsando la tecla 'r' del teclado se cambiará el estado del led rojo (de parpadear a no parpadear y viceversa), y que análogamente, con una pulsación de la tecla 'g' se cambiará el estado de parpadeo del led verde.