Curso de Linux Embebido
Segunda Parte

Libro de ejercicios prácticos

Carlos Guindel - Laboratorio de Sistemas Inteligentes

3 de julio de 2014

Sobre este documento

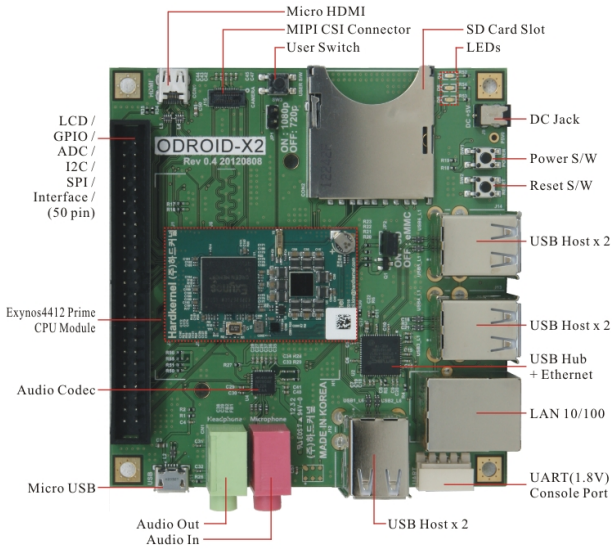
Este documento está basado en el trabajo de Free Electrons, disponible en http://free-electrons.com/doc/training/linux-kernel/ bajo los términos de Creative Commons CC BY-SA 3.0 license .

Copiar este documento

Información sobre la placa ODROID

Anotaciones sobre el ODROID

Documentación sobre la placa



La empresa Hardkernel ofrece información sobre la placa que estamos utilizando en `http://hardkernel.com/main/products/prdt_info.php?g_code=G135235611947&tab_idx=2`. No obstante, la principal fuente de información y soporte suelen ser los foros de ODROID `http://forum.odroid.com`.

La placa ODROID-X2 utiliza un SoC (*System-on-a-chip*) ARM Cortex-A9 MPCore, el Samsung Exynos4412 Prime, empleado en el Samsung Galaxy Note II, entre otros. La hoja de características de este chip está en su directorio de trabajo (`~/odroid-kernel`) bajo el nombre de `ExynosQuad.pdf`. Más adelante, necesitaremos recurrir a ella.

La frecuencia de reloj del microprocesador es de 1,7 GHz. El ODROID-X2 también incluye 2 GB de memoria RAM, 6 puertos USB 2.0 Host y un jack RJ-45 Ethernet 10/100 Mbps, entre otros.

Información sobre el arranque

En los dispositivos embebidos, es usual el uso de memorias flash que contienen el firmware necesario para el arranque y funcionamiento del sistema. Sin embargo, los ODROID únicamente disponen la tarjeta SD como medio de almacenamiento, de forma que todo el software, incluido el correspondiente al arranque, debe estar contenido en ella.[1].

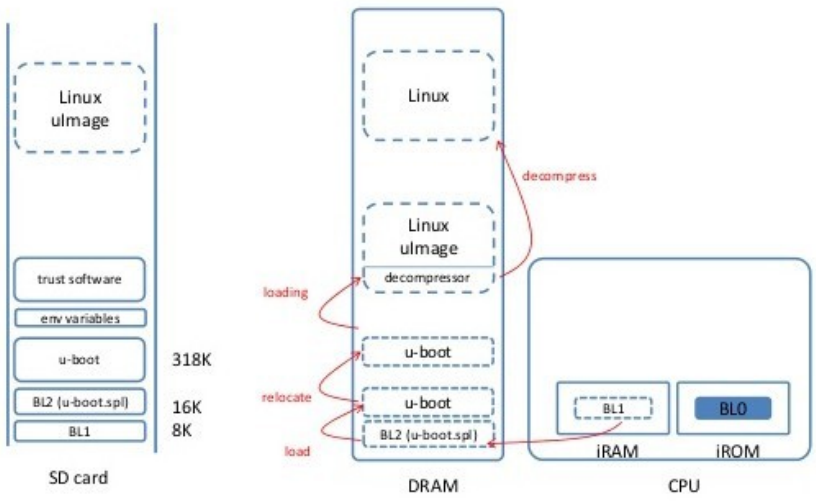
El *bootloader* que se utiliza es *Das U-Boot*, más conocido como U-Boot, que se ha convertido en un estándar de facto en este tipo de dispositivos, abarcando múltiples plataformas.

[1] También es posible el arranque desde eMMC mediante la modificación del *jumper* correspondiente, pero en este caso no se dispone de ella

La secuencia de arranque del SoC Exynos tiene ciertas particularidades impuestas por el fabricante Samsung. Una vista general de la secuencia de arranque que se desencadena con la conexión de la placa a la corriente es la siguiente:

1. iROM (código dentro del SoC) intenta leer el medio de arranque en sus primeros 512 bytes. Ahí debe de estar presente `fwbl1`.
2. `fwbl1` carga `bl2` que es parte del U-Boot.
3. `bl2` carga U-Boot.
4. `U-boot` realiza los procesos que faltan, como el manejo de TrustZone, la carga de la imagen del Kernel y Ramdisk (si procede).

Se trata de un arranque multietapa, ideado con el fin de inicializar incrementalmente más recursos en cada una de la etapas.



En resumen, en las primeras posiciones de la SD deben estar presentes:

- Posición 1. `fwbl1`. Es software propietario de Samsung, por lo que no se tiene demasiada información acerca de él
- Posición 31. `bl2`. Es el SPL, parte del U-Boot.
- Posición 63. `u-boot.bin` Es el U-Boot en sí mismo.
- Posición 2111. TrustZone Software. Es una extensión de seguridad para la arquitectura ARM.

Además de estos datos, necesarios en las fases tempranas del arranque, hay dos particiones más. A partir de la posición 3072, hay una partición FAT32 de 64 Mb, llamada `boot`, que contiene el kernel de Linux y el `initrd`. Por último, el sistema de ficheros raíz se ubica en otra partición, en este caso ext4, que en la SD utilizada recibe el nombre `trusty`, haciendo referencia a la version de (L)Ubuntu con la que trabajamos (14.04 Trusty Tahr).

En el propio Lubuntu del ODROID, la partición con la imagen de Linux aparece montada, por defecto, en `/media/boot` y en `/boot`.

Descarga del código del kernel

Creación de una copia propia de un árbol de Linux

Preparación

Vaya al directorio `~/odroid-kernel`.

Configuración de Git

El paquete `git` ya está instalado en el ODROID. En caso de que no lo estuviera, sería necesario instalarlo:

```
sudo apt-get install git
```

Antes de usar `git`, conviene especificar el nombre y la dirección de correo electrónico:

```
git config --global user.name 'Nombre'
git config --global user.email yo@dominio.com
```

Esa información se almacenará en los *commits* que se suban a los repositorios. Es importante configurarlos adecuadamente a la hora de generar y enviar parches, especialmente.

Clonar el árbol de Linux

Para empezar a trabajar con las fuentes del kernel de Linux, necesitaremos clonar su árbol git. El árbol de referencia es el mantenido por Linus Torvalds; sin embargo, los desarrolladores de ODROID (Hardkernel) tienen su propio árbol, generado a partir del principal, que será el que usemos.

En cualquiera de los dos casos, el código fuente ocupa casi 1 GB. En una situación normal, se podría clonar el árbol a través de Internet con:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

o, si el puerto para `git` está bloqueado, mediante el protocolo `http` con:

```
git clone http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

En el caso específico del árbol de Hardkernel, propio de los ODROID, habría que ejecutar (solo la rama 3.8.y):

```
git clone https://github.com/hardkernel/linux.git -b odroid-3.8.y odroid-3.8.y
```

Por limitaciones de tiempo, el árbol de Linux `odroid-3.8.y` ya está clonado en el sistema de ficheros del ODROID, dentro de la carpeta `~/odroid-kernel`.

Si los ODROID disponen de conexión a Internet, se pueden obtener las últimas actualizaciones realizadas desde el momento en que se clonó el árbol:

```
cd odroid-3.8.y
git checkout odroid-3.8.y
git pull
```

Aunque no resulta de utilidad para estos ejercicios, es posible comprobar el resto de ramas que existen en el repositorio:

```
git branch -a
```

Si hubieramos ejecutado `git clone` en el momento, el árbol obtenido ya habría estado actualizado, por lo que no sería necesario ejecutar `git pull`. No obstante, es posible ejecutar `git pull` siempre que se requiera actualizar la copia del árbol.

Compilación del kernel

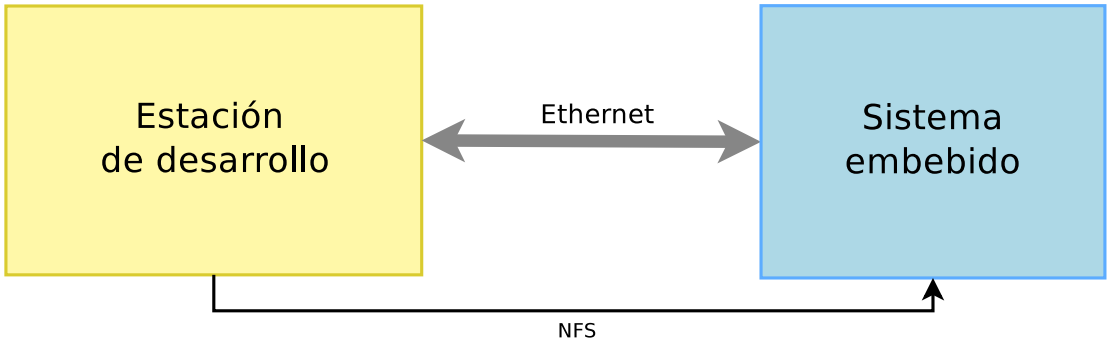
Objetivo: compilar el kernel para el ODROID

Despues de este ejercicio, deberá ser capaz de:

- Recompilar y cargar el kernel de Linux

Consideración en el desarrollo para sistemas embebidos

Es común, cuando se trabaja con Linux para sistemas embebidos, utilizar una estación de desarrollo aparte (que puede ser cualquier ordenador) para escribir y compilar el código, limitándose el uso del sistema embebido para las pruebas del mismo, como se muestra en la siguiente figura:



Sin embargo, la elevada potencia de procesamiento del ODROID-X2 permite su uso también en las tareas de desarrollo, lo cual resulta especialmente ventajoso desde el punto de vista de la complejidad de la instalación a realizar y del número de componentes necesarios.

No obstante, si se trabajara sobre una estación de trabajo distinta del ODROID, podría llevarse a cabo la compilación cruzada de todo tipo de programas mediante el uso de la *toolchain* adecuada:

```
cd <path-desarr.>
wget odroid.in/guides/ubuntu-lfs/arm-unknown-linux-gnueabi.tar.xz
tar -Jxf arm-unknown-linux-gnueabi.tar.xz
```

Previamente al inicio de la compilación del kernel, sería necesario fijar las siguientes variables de entorno:

```
export ARCH=arm
export CROSS_COMPILE=\
<path-desarr.>/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-
```

Configuración del kernel

La primera tarea a realizar es configurar el kernel con la configuración predefinida para el ODROID-X2. Dado que se está compilando sobre la propia plataforma, en este caso no es necesario fijar las variables de entorno `ARCH` y `CROSS_COMPILE`.

Para saber el nombre del archivo de configuración correspondiente, se puede:

- Ejecutar `make help`
- Comprobar la lista de archivos de configuración predefinidos disponibles en `arch/<arch>/configs/`, siendo `<arch>` la arquitectura correspondiente al microprocesador sobre el que se trabaja.

Una vez se conoce el nombre de la configuración predefinida, se debe usar adecuadamente el comando `make` para cargarla.

A modo de ejercicio, se plantea indagar si la configuración que se ha cargado (cuidado, no nos referimos al archivo presente en `arch/<arch>/configs/`) soporta el driver SPIDEV (para el puerto SPI - *Serial Peripheral Interface*), y si lo hace como módulo o integrado en el kernel.

Compilación y carga del Kernel

Compile el kernel. Para optimizar el proceso, utilice la opción `-j8` en el comando correspondiente, de forma tal que se utilicen todos los hilos de ejecución del microprocesador.

Habitualmente, la compilación del kernel es una tarea computacionalmente costosa, que conlleva un cierto tiempo, incluso en equipos de escritorio. No obstante, la operación ya ha sido llevada a cabo anteriormente en su ODROID y, además, el comando `make` compila únicamente aquellos ficheros que hayan sido modificados desde la última vez. Como en este caso no se ha modificado ninguno, el proceso es prácticamente inmediato.

Una vez finalizado el proceso, la imagen del kernel de Linux estará disponible en `arch/<arch>/boot/`. Para instalar el nuevo kernel recién compilado, basta copiarlo (con el comando `cp`) a la partición correspondiente de la tarjeta SD, sobrescribiendo el existente con el mismo nombre.

También pueden instalarse, en este momento, los módulos que se han compilado junto al kernel:

```
sudo make modules_install
```

Ahora, es necesario reiniciar para que se cargue el nuevo kernel. Si ha completado el proceso satisfactoriamente, notará que el led D6 del ODROID ha dejado de parpadear.

Escribir módulos

Objetivo: crear un módulo del kernel muy simple

Después de este ejercicio, deberá ser capaz de:

- Compilar y probar módulos del kernel independientes, cuyo código esté fuera de las fuentes de Linux.
- Escribir un módulo del kernel con ciertas capacidades, incluyendo el uso de parámetros.
- Acceder al interior del kernel desde el módulo.
- Compilar el módulo.

Escribiendo un módulo

En primer lugar, acceda a la carpeta `hola`, dentro del directorio de trabajo `odroid-kernel`, y examine el contenido de la misma.

Añada código C al fichero `hola_version.c` para implementar un módulo que muestra este mensaje al ser cargado:

```
Hola. Corriendo Linux <versión>.
```

... y muestre un mensaje de despedida cuando es descargado.

Para mostrar la versión de Linux, basta utilizar uno de los campos de la estructura `utsname()`. Utilice el siguiente comando en la carpeta con las fuentes de Linux (`odroid-3.8.y`) para comprobar como se usa esta estructura en otros archivos que ya forman parte de Linux.

```
git grep 'utsname'
```

Compilando el módulo

El directorio actual contiene un fichero `Makefile` que permite compilar módulos fuera del árbol del kernel. Compile su módulo.

Probando el módulo

Cargue el módulo. Para ello, se recomienda usar `insmod` con permisos de root. Los mensajes del módulo se deberían registrar en el log del kernel, accesible con `dmesg`.

Compruebe que funciona como debería y, si no, descárguelo, modifíquelo y compílelo las veces que sea necesario hasta que funcione.

Ejecute un comando para comprobar que su módulo está en la lista de módulos cargados. Ahora intente conseguir la lista de módulos cargados usando solo el comando `cat`.

Añadiendo un parámetro al módulo

Añada un parámetro llamado `quien` al módulo, de forma que pase a decir `Hola <quien>` en vez de `Hola`.

Compile y pruebe su módulo comprobando que tiene en cuenta el parámetro `quien` cuando es cargado.

Añadiendo información temporal

Mejore su módulo, de forma tal que cuando sea descargado, diga cuantos segundos pasaron desde que se cargo. Puede usar la función `do_gettimeofday()` para lograrlo.

Deberá buscar otros drivers en el código fuente del kernel que usen esa función.

Siguiendo los estándares de codificación de Linux

Si se pretende incluir algún día el módulo dentro del código fuente de Linux mantenido por Linus Torvalds, de forma tal que pueda usarse en todo el mundo, este deberá atenerse a unas ciertas normas de estilo.

Para asegurarse de cumplirlas, puede ejecutarse:

```
~/odroid-kernel/odroid-3.8.y/scripts/checkpatch.pl --file --no-tree hola_version.c
```

Si hay errores relativos a la indentación, se puede pasar el siguiente comando:

```
sudo apt-get install indent
indent -linux hola_version.c
```

Añadiendo el módulo hola_version a las fuentes del kernel

Como se van a hacer cambios en las fuentes del kernel, cree primero una rama especial para estos cambios:

```
git checkout odroid-3.8.y
git checkout -b hola
```

Añada los ficheros del módulo al directorio `drivers/misc/` dentro del árbol de Linux. Deberá modificar igualmente la configuración del kernel y los ficheros de compilación adecuadamente, de forma que se pueda seleccionar el módulo mediante `make xconfig` y compilarlo con el comando `make`.

Una vez comprobado que todo funciona correctamente, incorpore los cambios a la rama actual (intente añadir un mensaje adecuado):

```
cd ~/odroid-kernel/odroid-3-8-y/
git add -A
git commit -as
```

- `git add -A` añade (o borra) archivos al siguiente *commit*.
- `git commit -a` crea un *commit* con todos los ficheros modificados (que estén bajo seguimiento del repositorio) desde el *commit* anterior.
- `git commit -s` añade una firma al mensaje de *commit*. Las contribuciones al kernel de Linux deben disponer de esa línea.

Crear un parche del kernel

Si se desea compartir el módulo con otros usuarios, conviene crear un parche que añade los nuevos ficheros al árbol fuente del kernel. Se puede generar a partir de los cambios entre la rama actual y otra rama:

```
git format-patch odroid-3.8.y
```

Observe el archivo generado. Puede comprobar como su nombre ha utilizado el mensaje del *commit*. Si quiere cambiar el último mensaje del *commit* en este momento, puede ejecutar:

```
git commit --amend
```

Y volver a ejecutar `git format-patch` otra vez.

Acceder a la memoria I/O y los puertos

Objetivo: leer/escribir datos desde/a un dispositivo hardware

A través de los siguientes ejercicios, vamos a implementar un driver que permita manejar los leds del ODROID como si de un dispositivo orientado a caracteres se tratara.

Después de este laboratorio, deberá ser capaz de:

- Instanciar dispositivos de plataforma (*platform devices*) para la placa ODROID.
- Acceder a los registros E/S para controlar los leds.

Preparación

Vaya al directorio con el código fuente del kernel.

Cree una nueva rama para esta serie de ejercicios.

```
git checkout odroid-3.8.y
git checkout -b leds
```

Añadir dispositivos

La manera actual de describir los dispositivos de los que consta una placa de desarrollo, como el ODROID, es mediante un árbol de dispositivos (*Device Tree*). Esta forma de organización se ha introducido hace relativamente poco para la arquitectura ARM, por lo que Hardkernel todavía no soporta oficialmente esta característica. En vez de eso, los dispositivos se describen instanciándolos directamente como `struct platform_device` en código específico de la placa.

En el caso de los ODROID, este código se encuentra en `~/odroid-kernel/odroid-3.8.y/arch/arm/mach-exynos/mach-hkdk4412.c`.

En el código fuente del kernel presente en los ODROID, los led no aparecen como dispositivos. Ese es el motivo por el que, cuando lo compilamos y cargaron, estos dejaron de funcionar: en este momento, los pines correspondientes funcionan como entradas, pues ese es el valor de reset de los registros correspondientes en el microprocesador.

Ahora, vamos a escribir el código necesario para declarar la existencia de los leds de la placa. Será necesario declarar una estructura `struct platform_device` en la cual se inicialicen los campos siguientes:

```
static struct platform_device leds_curso = {
    .name = "led-curso",
    .id = -1,
    .num_resources = ARRAY_SIZE(led_curso_resources),
    .resource = led_curso_resources,
}
```

Para ello, anteriormente (deberá declararse antes en el código fuente), se tiene que crear la estructura `struct resource`

```
static struct resource led_curso_resources[] = {
    [0] = {
        .start = <dir-inicio-memoria>
        .end = <dir-fin-memoria>,
        .flags = IORESOURCE_MEM,
    },
};
```

En la primera estructura, se está informando de la existencia de un dispositivo que estará controlado por el driver `led-curso`. A este dispositivo, se le asocian como recursos hardware unas direcciones de memoria, que son las que se especifican en la segunda estructura, y se le pasan dentro del campo `resource`.

De acuerdo a las necesidades del dispositivo a controlar, podría especificarse otra información relevante para el driver a través del campo `platform_data`, dentro de la estructura `dev` que, a su vez, es un campo de `struct platform_device`.

Por otro lado, también se pueden asignar más recursos hardware al driver, tales como interrupciones (`IORESOURCE_IRQ`).

Con el fin de asignar adecuadamente las direcciones de memoria, es importante notar que **los leds de la placa están conectados a los pines GPC1[0] y GPC1[2]** del SoC Exynos. Asigne como recurso correspondiente a los leds todo el conjunto de direcciones de memoria de los registros correspondientes a los pines GPC1. Para ello, use la hoja de características del SoC. Hay que tener en cuenta que, de esta forma, estamos considerando los dos leds como un mismo dispositivo.

Una vez modificado convenientemente el fichero `mach-hkdk4412.c`, compile de nuevo el kernel, copie la imagen a la partición correspondiente de la tarjeta SD y reinicie la placa.

Operar un *platform driver*

En `~/odroid-kernel/drivers` puede encontrar un archivo `led_curso.c` con un esqueleto de driver para un dispositivo de plataforma (*platform driver*).

Añada la información necesaria para asociar el driver con el dispositivo que ha declarado en el fichero `mach-hkdk4412.c`.

Compile el módulo y cárguelo. Compruebe los mensajes del núcleo, que deberían confirmar que se ha llamado la rutina `probe()`. Si no se ha producido la llamada, no se ha llevado a cabo correctamente la asociación entre driver y dispositivo y hay que revisar los códigos anteriores.

Obtener la dirección base de memoria desde el driver

Vamos a leer de registros de memoria mapeados y a escribir sobre ellos. Lo primero que necesitamos es la dirección física base para el dispositivo.

Esta información está disponible como recurso del driver. Se puede extraer con:

```
struct resource *res;
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

Añada este código a su rutina `probe()`, con el manejo de error adecuado cuando `res == NULL`, e imprima la dirección de inicio (`res->start`) para asegurarse de que los valores de

dirección que ha obtenido coinciden con los esperados.

En cuanto obtenga valores correctos, puede eliminar la instrucción de imprimir.

Crear una estructura privada de dispositivo

El siguiente paso es empezar a asignar y registrar recursos, los cuales tendrán que ser eventualmente liberados y desregistrados, también.

Necesitamos crear una estructura que contenga información específica del dispositivo y ayude a mantener punteros entre dispositivos lógicos y físicos.

La idea es que el mismo driver pueda servir para controlar distintos dispositivos. Por ello, no es posible utilizar variables globales que sean accesibles desde las rutinas `probe` y `remove` al mismo tiempo, y se usa esta estructura *privada* para gestionar toda la información referente al dispositivo en concreto.

Lo primero que almacenaremos en esta estructura es la dirección virtual base para cada dispositivo (obenida a través de `ioremap()`), declaremos esta estructura como sigue:

```
struct led_curso_dev {
        void __iomem *regs;
};
```

Ahora, hay que reservar memoria para esa estructura al principio de la rutina `probe()`, mediante la función `devm_kzalloc`. De este modo, la memoria se libera automáticamente cuando deje de ser necesario.

Por tanto, añada la siguiente línea al código:

```
struct led_curso_dev *dev;
...
dev = devm_kzalloc(&pdev->dev, sizeof(struct feserial_dev), GFP_KERNEL);
```

Ahora, podemos obtener una dirección virtual para la dirección base física del dispositivo, llamando a:

```
        dev->regs = devm_request_and_ioremap(&pdev->dev, res);

        if (!dev->regs) {
                dev_err(&pdev->dev, "Cannot remap registers\n");
                return -ENOMEM;
        }
```

Lo interesante es que no será necesario liberar este recurso, ni en la rutina `remove()`, ni en el caso de que haya errores en los siguientes pasos de la rutina `probe()`.

Asegúrese de que su driver actualizado compila, se carga y se descarga correctamente.

Inicialización del dispositivo

Ahora que tenemos una dirección virtual para acceder a los registros, estamos preparados para configurar unos cuantos de ellos para configurar adecuadamente los led. Por supuesto, esto se hace en la rutina `probe()`.

Acceso a los registros del dispositivo

Para facilitar la lectura de los registros, cree una rutina `reg_read()` que devuelva un valor `unsigned int` y tome como argumentos un puntero `dev` a una estructura `led_curso_dev` y un offset entero `offset`.

En esta función, lea de un registro de 32 bit en la dirección virtual base del dispositivo más un cierto offset.

Cree una rutina `reg_write()` similar, escribiendo un valor entero sin signo en un cierto offset entero desde la dirección virtual base del dispositivo. El prototipo debería parecerse a:

```
static void reg_write(struct led_curso_dev *dev, int val, int off);
```

Ya estamos listos para leer y escribir registros.

Configuración de los leds como salidas

Utilice las funciones anteriores para configurar los pines correspondientes a los led como salidas, durante la rutina `probe()`. Para ello, ayúdese de la documentación del SoC Exynos.

Pruebe, del mismo modo, a escribir en los registros que controlan el encendido y apagado de los leds durante la rutina `probe()`.

En todos estos casos, puede ser útil el uso del macro `BIT()`, que proporciona una máscara para un bit concreto.

Depuración del driver

Descargue el módulo e intente cargarlo de nuevo. Si el segundo intento de cargar el módulo falla, probablemente se deba a que su driver no libera adecuadamente los recursos que reservó, ya sea en el momento de salir del módulo, o cuando se produce algún fallo durante su funcionamiento. Revise y arregle su módulo si experimenta estos problemas.

Driver *misc*

Objetivo: impementar las operaciones de archivo de un driver misc

Después de este ejercicio, deberá ser capaz de:

- Escribir un driver *misc* simple, permitiendo simular la lectura y escritura de caracteres mediante el uso de los leds.
- Escribir funciones simples de tipo `file_operations` para un dispositivo, incluyendo controles `ioctl`.
- Copiar datos desde el espacio de memoria del usuario hacia el espacio de memoria del kernel y, eventualmente, al dispositivo.

Registro del driver misc

Nos proponemos dotar de una interfaz al driver de los leds. Para ello, usaremos el *framework Misc* para implementar un driver simple. La idea es controlar los leds mediante el envío de caracteres. Por ello, los fundamentos en los que nos vamos a basar son exactamente los mismos que se podrían usar, por ejemplo, en el desarrollo de un driver para comunicación por el puerto serie, ya que, en realidad, se efectuará el envío y recepción de caracteres.

Empecemos añadiendo la infraestructura para registrar un driver *misc*.

Lo primero que hay que hacer es crear:

- Una función de escritura de ficheros `led_curso_write()`. Por ahora, coloque únicamente `return -EINVAL;` en la función para señalar que aún hay algo incorrecto en esa función.
- De forma similar, una función `led_curso_read()` para la lectura de ficheros.
- Una estructura `file_operations` declarando las dos operaciones sobre ficheros anteriores.

El siguiente paso es crear una estructura `miscdevice` e inicializarla. Sin embargo, nos enfrentamos, como es habitual, a la restricción para manejar múltiples dispositivos. Tenemos que añadir una estructura como la siguiente a nuestra estructura privada del dispositivo:

```
struct led_curso_dev {
        struct miscdevice miscdev;
        void __iomem *regs;
};
```

Para poder acceder a nuestra estructura privada de datos en otras partes del driver, es necesario adjuntarla a la estructura `pdev` usando la función `platform_set_drvdata()`. Explore el código fuente para saber cómo se hace en otros ejemplos.

Ahora, al final de la rutina codeprobe(), cuando el dispositivo está totalmente preparado para funcionar, puede inicializar la estructura `miscdevice` para cada dispositivo encontrado, de forma tal que:

- Obtenga un número menor automáticamente asignado.

- Especifique un nombre para el fichero del dispositivo en *devtmpfs*. Proponemos el uso de `kasprintf(GFP_KERNEL, "led-curso-\%x", res->start)`. `kasprintf()` reserva memoria para un buffer y ejecuta `ksprintf()` para rellenar su contenido. No olvide llamar a `kfree()` en este buffer en la función `remove()`.
- Pase la estructura de operaciones de ficheros que definió anteriormente.

Lo último que hay que hacer (al menos para tener un driver *misc*, aun cuando las operaciones de ficheros no están listas todavía), es añadir las rutinas de registro y desregistro. Suele ser ahora cuando se necesita acceder a la estructura `led_curso` para cada dispositivo desde la estructura `pdev` que se pasa a la rutina `remove()`.

Asegúrese de que su driver compila y carga correctamente, y de que puede ver un archivo de dispositivo en `/dev`.

En este momento, asegúrese de que puede cargar y descargar el driver varias veces. Esto suele revelar problemas en el registro y el desregistro, en caso de que estos existan.

Implementar rutina de apertura de fichero

El siguiente paso es implementar la rutina `write()`. Sin embargo, necesitaremos acceder a nuestra estructura `led_curso_dev` desde dentro de esa rutina.

Por el momento, es necesario implementar una operación de apertura de fichero para adjuntar una estructura privada a un archivo de dispositivo abierto. En realidad, esa función no va a hacer nada, pero es necesaria para el correcto funcionamiento del driver.

Implementar rutina write()

Ahora, añada código a su función de escritura, para interpretar los datos enviados por el usuario (en forma de caracteres) a los leds.

Lo primero que hay que hacer es recuperar la estructura `led_curso_dev` desde la estructura `miscdevice` en sí misma, accesible a través del campo `private_data` de la estructura de apertura de fichero (`file`).

En el momento en que registramos nuestro dispositivo *misc*, no mantuvimos ningún puntero a la estructura `led_curso_dev`. Sin embargo, la estructura `miscdevice` es accesible, y al ser un miembro de la estructura `led_curso_dev`, podemos usar una macro "mágica" para obtener la dirección de la estructura padre:

```
struct feserial_dev *dev =
container_of(file->private_data, struct led_curso_dev, miscdev);
```

Ahora, añada código para hacer que los leds representen en binario el valor decimal enviado por el usuario. Evidentemente, se espera que el valor decimal enviado por el usuario deberá estar entre 0 y 3; puede especificar el comportamiento que desee en caso contrario.

Una vez lo haya hecho, compile y cargue su módulo. Pruebe que su función `write` funciona correctamente usando:

```
echo '1' | sudo tee -a /dev/led-curso-1140080
```

Deberá producirse el comportamiento de los leds que se ha especificado en el driver (canónicamente, encenderse uno de ellos).

Implementar rutina read()

Vamos a implementar también una rutina simple de lectura, para que el usuario pueda leer el estado actual de los leds.

Para ello, en primer lugar, tiene que obtener la estructura privada del dispositivo, de forma idéntica a como se hizo en la rutina `write()`. Con ella disponible, puede acceder a los registros correspondientes al estado de los leds.

Lo más importante en este caso es poner a disposición del usuario los datos obtenidos, para lo cual es necesario usar una de las funciones específicas que relacionan el espacio del kernel y el del usuario.

Dado que conviene mostrar el resultado en forma de carácter, se recomienda convertir el entero que se va a obtener al leer los registros:

```
unsigned char led_char = (char)(((int)'0')+led_int);
```

Para comprobar el funcionamiento de esta función, se puede usar:

```
sudo cat /dev/led-curso-1140080
```

El problema, como se puede comprobar, es que el valor se imprime continuamente. Conviene, por tanto, ajustar el offset dentro de `read` correctamente.

```
if(*off > 0)
     return 0; /* Fin de fichero */

/* Copia de datos al usuario */

*off+=1;
return 1;
```

Ejercicio opcional: ioctl

Queremos mantener una cuenta del número de veces que se ha cambiado el estado de los leds. Por tanto, necesitamos implementar dos operaciones `unlocked_ioctl()`:

- `LED_RESET_COUNTER`, que resetee el contador.
- `LED_GET_COUNTER`, que devuelva el valor actual del contador en una variable pasada por dirección.