_ TELECOM ESCUELA
TÉCNICA VLC SUPERIOR
DE INGENIERÍA DE
TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Desarrollo de drivers para acceso a bajo nivel a los periféricos de un sistema linux embebido

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Liu , Cai

Tutor/a: Torres Carot, Vicente

CURSO ACADÉMICO: 2021/2022

Resumen

Este trabajo presenta cómo diseñar, desarrollar y testear un driver de dispositivo de caracteres en sistemas Linux embebido. Contiene muchos conceptos básicos sobre el desarrollo de drivers, incluido cómo registrar un dispositivo en el kernel, cómo obtener información del dispositivo, cómo utilizar los diferentes registros de control para operar el dispositivo, etc.

Hay 3 métodos de desarrollo incluidos en este trabajo: operar directamente los registros de control en el código fuente del driver desarrollado, emplear el modelo bus-driver-dispositivo para implementar dispositivos de plataforma y drivers de plataforma, y agregar nodos de dispositivos al árbol de dispositivos, accederlos y operarlos en el programa de driver.

En este proyecto se utiliza la placa de desarrollo STM32MP157F-DK2 que se basa en una arquitectura de procesamiento multinúcleo heterogénea con Cortex-A7 y Cortex-M4, utilizando los paquetes de software Starter Package y Developer Package proporcionados oficialmente por STMicroelectronics para crear una distribución embebida de Linux para STM32MPU.

Resum

Aquest treball presenta com dissenyar, desenvolupar i testejar un driver de dispositiu de caràcters en sistemes Linux embegut. Conté molts conceptes bàsics sobre el desenvolupament de drivers, inclòs com registrar un dispositiu al nucli, com obtenir informació del dispositiu, com utilitzar els diferents registres de control per operar el dispositiu, etc.

Hi ha 3 mètodes de desenvolupament inclosos en aquest treball: operar directament els registres de control al codi font del driver desenvolupat, emprar el model bus-driver-dispositiu per implementar dispositius de plataforma i drivers de plataforma, i afegir nodes de dispositius a l'arbre de dispositius, accedir-los i operar-los al programa de driver.

En aquest projecte es fa servir la placa de desenvolupament STM32MP157F-DK2 que es basa en una arquitectura de processament multi-nucli heterogènia amb Cortex-A7 i Cortex-M4, utilitzant els paquets de programari Starter Package i Developer Package proporcionats oficialment per STMicroelectronics per crear una distribució embeguda de Linux per a STM32MPU.

Abstract

This thesis presents how to design, develop and test a character device driver on embedded Linux systems. It contains many basics about driver development, including

how to register a device in the kernel, how to get device information, how to use the different control registers to operate the device, etc.

There are 3 development methods included in this work: directly operating the control registers in the source code of the developed driver, using the bus-driver-device model to implement platform devices and platform drivers, and adding device nodes to the device tree. devices, access and operate them in the driver program.

This project uses the STM32MP157F-DK2 development board which is based on a heterogeneous multi-core processing architecture with Cortex-A7 and Cortex-M4, using the Starter Package and Developer Package software officially provided by STMicroelectronics to create an embedded Linux distribution for STM32MPU.

Índice

Capítulo 1. Introducción y objetivos 1
Capítulo 2. Preparaciones 3
2.1 Instalación de Linux embebido en STM32MP157F-DK2 3
2.2 Entornos y herramientas 4
2.3 Configuración de la conexión a internet 4
Capítulo 3. Conceptos 6
3.1 El papel de los Drivers 6
3.2 Clasificación de drivers 6
3.2.1 Dispositivos de carácter 7
3.2.2 Dispositivos de bloque 7
3.2.3 Dispositivos de red 7
Capítulo 4. Implementación de un driver de carácter de forma simple 8
4.1 Introducción 8
4.2 La estructura file_operations 9
4.2.1 Control de GPIO en STM32MP157 10
4.2.2 Gestión de memoria 14
4.3 Número de dispositivo 15
4.3.1 Definición 16
4.3.2 Configuración 16
4.4 Registro del dispositivo 17
4.5 Nodo de dispositivo 18
4.5.1 Creación y eliminación de clase 18
4.5.2 Creación y eliminación de dispositivo 18
4.6 Archivos de cabecera y licencias 19
4.7 Carga y descarga del driver 19
4.8 Aplicación de verificación y Makefile 20
4.9 Resultados 22
Capítulo 5. Implementación de dispositivos y drivers de plataforma 25
5.1 Modelo de bus - driver - dispositivo 25
5.2 Introducción a dispositivos y drivers de plataforma 26
5.3 Dispositivo de plataforma 26
5.3.1 Definición de recursos 27

5.3.2 Registro de un dispositivo de plataforma 28
5.4 Driver de plataforma 28
5.4.1 Obtención de recursos 29
5.4.2 Registro de un driver de plataforma 29
5.5 Resultados 30
Capítulo 6. Árbol de dispositivos 32
6.1 Introducción 32
6.2 Sintaxis del árbol de dispositivos 32
6.2.1 Ficheros de cabecera 33
6.2.2 Nodo de dispositivo 34
6.2.3 Propiedades de nodo 34
6.3 Crear un nodo en el árbol de dispositivos 36
6.4 Realización del driver empleando Árbol de dispositivos 37
6.4.1 Funciones _of 37
6.4.2 Emparejamiento 39
6.5 Resultados 39
Capítulo 7. Conclusiones y propuesta de trabajo futuro 41
Capítulo 8. Bibliografía 42
Anexo I. Código fuente del proyecto realizado en el capítulo 4 44
Anexo II. Código fuente del proyecto realizado en el capítulo 5 50
Anexo III. Código fuente del proyecto realizado en el capítulo 6 59
Anexo IV. Código fuente de la aplicación de verificación 67

Capítulo 1. Introducción y objetivos

Desde los teléfonos móviles hasta el campo de la conducción no tripulada, los sistemas embebidos se utilizan ampliamente en nuestra vida diaria. Un dispositivo embebido es un dispositivo que integra el software y hardware que puede realizar varias funciones, como la interacción persona-computadora, recopilar datos de sensores, procesar datos y realizar comunicaciones. Además, tiene las características de tamaño pequeño, bajo consumo de energía, bajo costo, rendimiento alto en tiempo real y alta confiabilidad.

El objetivo de este TFG es estudiar las opciones existentes para el acceso a bajo nivel a los periféricos en un sistema Linux embebido, desarrollando una aplicación que demuestre el funcionamiento de la solución desarrollada.

En este TFG se utiliza el dispositivo embebido STM32MP157F-DK2 (*STM32MP157F, Discovery kit*) fabricado por la compañía STMicroelectronics. Viene con un microprocesador de la serie STM32MP1, que se basa en una arquitectura de procesamiento multinúcleo heterogénea que consta de Cortex-A7 y Cortex-M4, lo que brinda soporte para el sistema operativo de código abierto Linux y soporte para el ecosistema de MCU STM32, respectivamente, facilita el desarrollo de aplicaciones más amplias.

El objetivo principal de cada capítulo se describe a continuación:

Capítulo 1. Introducción del proyecto.

Capítulo 2. Preparar las herramientas necesarias y el entorno de desarrollo, crear una imagen de la distribución STM32 MPU OpenSTLinux proporcionada por STMicroelectronics oficialmente e instalarlo en nuestra placa de desarrollo STM32MP157F-DK2.

Capítulo 3. Introducir los conceptos básicos de drivers y obtener una idea general de los distintos tipos de dispositivos en un sistema Linux embebido.

Capítulo 4. Implementar un driver de LED de la manera más básica para realizar la función de encendido y apagado del LED. Al mismo tiempo, presentar todo el proceso de control de un LED en la placa STM32MP157F, así como los procesos de desarrollo de un driver. Verificar el driver implementado una aplicación y mostrar los resultados obtenidos.

Capítulo 5. Aprender el modelo bus-driver-dispositivo en el sistema Linux, estudiar la forma de implementar y agregar un dispositivo de plataforma al kernel, realizar un driver de plataforma correspondiente para controlar el dispositivo. De modo similar, verificar y mostrar resultados.

Capítulo 6. Comprender el papel del árbol de dispositivos en el sistema Linux embebido y su estructura. Agregar un nodo de dispositivo en el árbol de dispositivos y desarrollar un driver a través de funciones definidas en kernel para realizar operaciones sobre los

nodos en el árbol de dispositivo. Comprobar que el nodo se ha añadido correctamente al kernel y verificar el driver desarrollado.

Capítulo 7. Sugerir posibles mejoras y trabajos futuros.

Capítulo 8. Bibliografía de referencia.

Anexo I. Código fuente del proyecto realizado en el capítulo 4.

Anexo II. Código fuente del proyecto realizado en el capítulo 5.

Anexo III. Código fuente del proyecto realizado en el capítulo 6.

Anexo IV. Código fuente de la aplicación de verificación.

Capítulo 2. Preparaciones

2.1 Instalación de Linux embebido en STM32MP157F-DK2

Para simplificar el desarrollo sobre los dispositivos STM32 MPU. ST proporciona una distribución de software embebido STM32MPU (*STM32MPU Embedded Software*) que consta de software, creación de sistemas y herramientas de desarrollo. [1]

Se proporcionan tres paquetes con diferentes propósitos para esta distribución, que son el paquete de inicio (*Starter Package*), el paquete de desarrollador (*Developer Package*) y el paquete de distribución (*Distribution Package*). [2]

En este trabajo se utiliza el paquete de distribución para construir el sistema Linux embebido. Su instalación se basa en la instalación del paquete de inicio. Por ejemplo, parte de los contenidos que se proporcionan son [3]:

- Un kit de desarrollo de software (SDK) que contiene un compilador cruzado para nuestro Linux host.
- Código fuente de los siguientes softwares de la distribución OpenSTLinux:
  - U-Boot
  - Trusted Firmware-A (TF-A)
  - Linux kernel
  - Opcionalmente, entorno de ejecución de confianza de código abierto (OP-TEE)

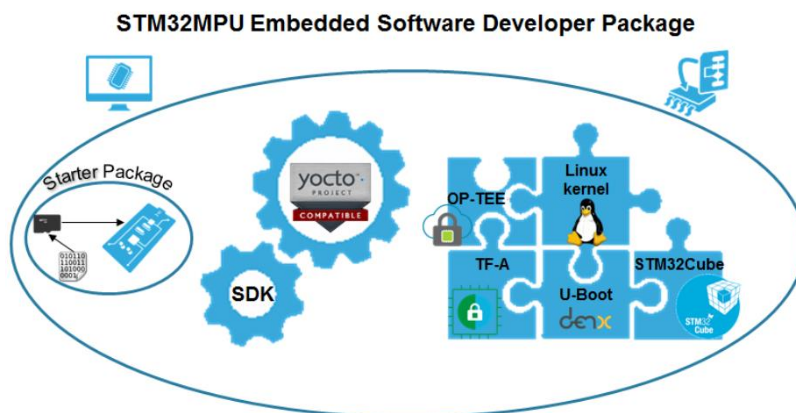

Figura 1. Contenido del paquete de desarrollador [3]

El método para instalarlo se describe en detalle en la wiki de ST.[1] Una vez finalizada la instalación, utilizaremos la herramienta STM32CubeProgrammer que se ha instalado junto con el paquete para instalar la imagen creada a la placa STM32MP157F-DK2.[2]

```
[  OK  ] Started Update UTMP about System Runlevel Changes.
ST OpenSTLinux - Weston - (A Yocto Project Based Distro) 3.1.11-openstlinux-5.10-dunfell-mp1-21-11-17 stm32mp1 ttySTM0
stm32mp1 login: root (automatic login)
```

Figura 2. Información que se muestra al arrancar el sistema

[1] https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package#

[2] https://wiki.st.com/stm32mpu/wiki/STM32MP15_Discovery_kits_-_Starter_Package#Image_flashing

2.2 Entornos y herramientas

Otras herramientas y entornos utilizados en el proceso de desarrollo son los siguientes:

Ubuntu: El desarrollo de Linux debe realizarse bajo el sistema Linux. Ubuntu es un sistema operativo Linux basado en aplicaciones de escritorio con características fáciles de usar. En este TFG se utiliza la versión 18.04 de Ubuntu.

Visual Studio Code: VSCode es un editor de código lanzado por Microsoft, que es gratuito, de código abierto y potente. Se utiliza en este proyecto para leer el código fuente del kernel de Linux y desarrollar los programas.

Makefile: Makefile describe las reglas de compilación de los proyectos C/C++ bajo el sistema Linux e indica el orden de compilación de los archivos fuente, las dependencias, etc. En este proyecto, vamos a escribir archivos Makefile para compilar los programas de driver y la aplicación.

MobaXterm: MobaXterm es un software de computación remota que admite SSH, FTP, SFTP, serie y otros protocolos. En el proceso de desarrollo, conectamos la placa de desarrollo a la computadora mediante un cable serie y usamos MobaXterm para conectar el puerto serie.

Compilador cruzado: El compilador *gcc* que viene con Ubuntu es para la arquitectura X86, el uso de compilador cruzado es imprescindible si queremos ejecutar el programa en la arquitectura ARM. Utilizaremos el compilador cruzado provisto en el kit de desarrollo de software (SDK) del paquete de desarrollador, los pasos de instalación se detallan en la wiki[3].

2.3 Configuración de la conexión a internet

Por lo general, necesitamos transmitir el programa de driver y la aplicación de verificación compilados en el sistema Linux a la placa de desarrollo para ejecutarse. STM32MP157F-DK2 incluye un módulo wifi, podemos conectarla a wifi para realizar la comunicación desde Linux a la placa sin conectar el cable de red. Los pasos para conectarse a wifi se describen en la wiki.[4]

Para verificar que la wifi está conectada correctamente, podemos enviar comandos ping entre la PC y la placa de desarrollo utilizando las direcciones IP.

[3] https://wiki.st.com/stm32mpu-ecosystem-v2/wiki/Getting_started/STM32MP1_boards/STM32MP157x-DK2/Develop_on_Arm%C2%AE_Cortex%C2%AE-A7/Install_the_SDK

[4] https://wiki.st.com/stm32mpu/wiki/How_to_setup_a_WLAN_connection

```
root@stm32mp1:~# ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:9D:6B:A5:58:A5
          inet addr:192.168.18.194  Bcast:192.168.18.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figura 3. Dirección IP de la placa

Los resultados se muestran en la figura 3 y figura 4.

```
root@stm32mp1:~# ping 192.168.18.193
PING 192.168.18.193 (192.168.18.193) 56(84) bytes of data.
64 bytes from 192.168.18.193: icmp_seq=1 ttl=64 time=28.5 ms
64 bytes from 192.168.18.193: icmp_seq=2 ttl=64 time=17.4 ms
64 bytes from 192.168.18.193: icmp_seq=3 ttl=64 time=25.6 ms
64 bytes from 192.168.18.193: icmp_seq=4 ttl=64 time=13.6 ms
64 bytes from 192.168.18.193: icmp_seq=5 ttl=64 time=21.6 ms
^C
--- 192.168.18.193 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 13.587/21.339/28.517/5.386 ms
```

Figura 4. Ping desde la placa al Linux host

```
cai@cai-virtual-machine:~$ ping 192.168.18.194
PING 192.168.18.194 (192.168.18.194) 56(84) bytes of data.
64 bytes from 192.168.18.194: icmp_seq=1 ttl=64 time=32.6 ms
64 bytes from 192.168.18.194: icmp_seq=2 ttl=64 time=19.3 ms
64 bytes from 192.168.18.194: icmp_seq=3 ttl=64 time=7.33 ms
64 bytes from 192.168.18.194: icmp_seq=4 ttl=64 time=32.2 ms
^C
--- 192.168.18.194 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 7.335/22.909/32.689/10.469 ms
```

Figura 5. Ping desde el Linux host a la placa

Cuando tenemos el internet conectado, podemos completar la configuración para poder transferir datos entre el host Linux y la placa de desarrollo mediante el protocolo SSH.[5] De esta forma, se puede utilizar el comando para realizar la copia de archivos:

```
scp root@<dirección IP de la placa>:/<ruta del archivo de la placa>/<ejemplo.txt> <ruta del archivo del host>/
```

[5] https://wiki.st.com/stm32mpu/wiki/How_to_transfer_a_file_over_network

Capítulo 3. Conceptos

3.1 El papel de los Drivers

En general, un sistema Linux embebido está compuesto por tres partes: software, sistema operativo y hardware.

La parte de software se refiere a aplicaciones específicas, los usuarios pueden diseñar el software de acuerdo con diferentes necesidades para lograr diferentes funcionamientos del dispositivo. En cambio, hardware es el conjunto del microprocesador y los periféricos integrados, es la plataforma en la que se puede ejecutar el software.

Para que el proceso de desarrollo de los dos sea independiente, en otras palabras, para poder diseñar el software sin conocer los detalles del hardware, y poder implementar el hardware sin especificar el tipo de aplicación, hacemos el uso de los drivers.

Los ficheros de driver están situados en el kernel de Linux en un sistema Linux embebido, contienen las informaciones del hardware y funciones para manejar directamente los periféricos a bajo nivel. De esta manera, en una aplicación podemos realizar el control del hardware a través de los drivers, nos permite leer y escribir en los registros de dispositivos de acuerdo con el funcionamiento específico del hardware. Se puede decir que los drivers proporcionan una interfaz entre la aplicación y el hardware para realizar la comunicación de datos entre ellos.

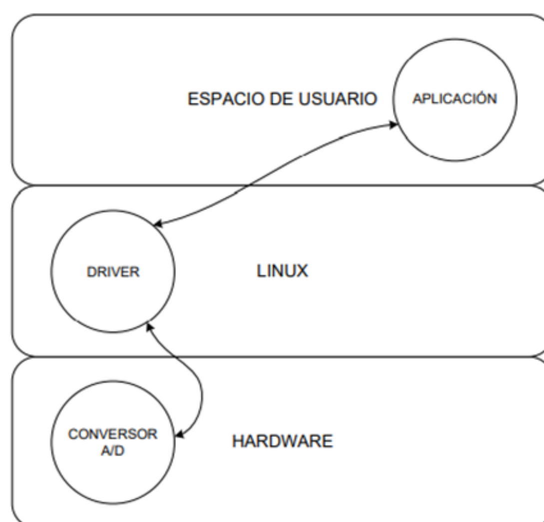

Figura 6. Estructura de un sistema Linux embebido [4]

3.2 Clasificación de drivers

En un sistema Linux, los dispositivos se puede dividir en tres categorías diferentes: Dispositivos de carácter, dispositivos de bloque y el módulo de red. [5]

3.2.1 Dispositivos de carácter

Hace referencia a los dispositivos que se acceden como flujos de bytes (archivos). En comparación con un archivo ordinario, se lee y se escribe en la secuencia de los datos de entrada en serie. Por ejemplo, los teclados, LED, LCD, etc. más comunes son todos dispositivos de caracteres.

3.2.2 Dispositivos de bloque

Los dispositivos de bloque típicos incluyen discos duros, tarjetas SD, memoria flash y otros dispositivos que acceden a la información en unidades de "bloques" de 512 bytes, 1024 bytes o 4096 bytes generalmente, y sobre los cuales se pueden construir un sistema de archivos. [6]

Los dispositivos de bloques y los dispositivos de caracteres difieren en la gestión interna del kernel. Cada operación de lectura y escritura en un dispositivo de caracteres pasará directamente al driver correspondiente en el kernel del sistema. Sin embargo, las operaciones sobre un dispositivo de bloque deben pasar por un búfer del sistema, y pasará indirectamente al driver.

3.2.3 Dispositivos de red

Los dispositivos de red son especiales en el sistema Linux, a diferencia de los dispositivos de caracteres y los dispositivos de bloques, que generalmente implementan operaciones como lectura y escritura, se encarga del intercambio de datos con otros hosts a través de interfaces.

Por tanto, según el dispositivo a controlar, podemos dividir los drivers en 3 tipos respectivamente: drivers de carácter, drivers de bloque y drivers de red.

Capítulo 4. Implementación de un driver de carácter de forma simple

En este capítulo, implementaremos un driver de LED de la manera más básica para realizar la función de encendido y apagado del LED.

Desde la manual de usuario de STM32MP157F y la descripción de hardware en la página web oficial de ST, vemos que tenemos 4 LED de usuario, y vamos a utilizar el LD5.

E/S	Configuración	Objetivo
PA14	PA14 está conectado al LED verde LD5, activa a nivel bajo.	Se puede utilizar en tiempo de ejecución para ejemplos de Linux.
PA13	PA13 está conectado al LED rojo LD6, activa a nivel bajo.	Se utiliza para mostrar la información de arranque de Cortex-A.
PH7	PH7 está conectado al LED naranja LD7, activa a nivel alto.	Indicación de entrada/salida de U-Boot y el latido de Linux, que parpadea mientras Linux esté activo en Cortex-A.
PD11	PD11 está conectado al LED azul LD8, activa a nivel alto.	Usado para el veredicto de ejemplos de STM32Cube en STM32MP15. Disponible para usuarios en STM32MP13.

Tabla 1. Configuración E/S de las interfaces de LED [7] [8]

4.1 Introducción

En el kernel de Linux, los dispositivos de carácter se definen mediante la estructura *cdev*, podemos decir que cada objeto *cdev* representa un dispositivo de carácter. En *cdev* están almacenado la información relevante del dispositivo de caracteres (número de dispositivo, *dev_t*), la apertura, lectura, escritura, cierre y otras interfaces de operación del dispositivo de carácter (*file_operations*). Cuando queremos agregar un dispositivo de caracteres en el sistema, registramos un nuevo objeto *cdev* en el kernel, y lo vinculamos con un archivo en la carpeta */dev* (nodo del dispositivo). Así que cuando leemos y escribimos este archivo, se encontrarán el dispositivo y sus interfaces de operación en el kernel a través del sistema de archivos virtual, y se operará directamente el dispositivo. [6]

Para empezar, definimos nuestra propia estructura *my_led_dev* para almacenar las informaciones necesarias para el desarrollo del driver.

```
struct my_led_dev{
        dev_t dev_nb;               // device number
        int major;                  // major device number
        int minor;                  // minor device number
        struct cdev cdev;           // cdev
        struct class *class;        // class
```

```
        struct device *device;    // device
};
struct my_led_dev my_led;
```

4.2 La estructura file_operations

Como hemos mencionado anteriormente, en un sistema Linux cada dispositivo de carácter está vinculado con un archivo en la carpeta */dev* dentro del Kernel, en concreto, un archivo llamado */dev/driver_name* (nombre del driver del periférico). De esta forma la aplicación puede manipular el periférico operando este archivo. Por ejemplo, se puede utilizar la función *open()* en una aplicación para abrir el dispositivo, utilizar la función *read()* para obtener información sobre el dispositivo, con la función *write()* se configura el estado del dispositivo y finalmente invocar la función *close()* para cerrar el dispositivo.

Sin embargo, la aplicación se ejecuta en el espacio del usuario, y las funciones *open()*, *read()*, *write()*, *open()* y otras que podemos aplicar dentro de una aplicación son funciones de operación de archivos proporcionadas por la librería C, y no pueden operar directamente el hardware. En este caso, se exige la ayuda del mecanismo de llamada al sistema (en inglés *system call*) para invocar las funciones correspondientes de abrir, leer, escribir y cerrar proporcionadas en el driver, para realizar controles directos del hardware. La figura 7 muestra el diagrama de flujo de la operación del hardware por una aplicación de Linux:

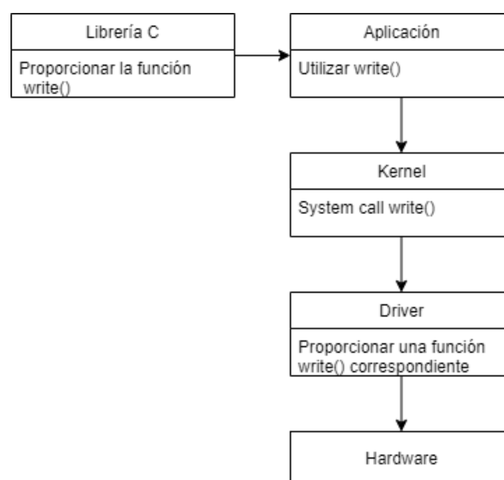

Figura 7. Proceso de operación del hardware por una aplicación de Linux

En síntesis, para cada función utilizada por una aplicación con el objetivo de manipular un dispositivo, existe una función driver correspondiente descrito en su fichero de driver. Dentro del archivo *include/linux/fs.h* en el kernel de Linux hay una estructura denominada *file_operations*, que es la colección de todas las funciones de operación en los drivers del kernel de Linux.

En resumen, para implementar la funcionalidad de encender y apagar un led, la idea principal sería construir nuestra propia estructura *file_operations* que contiene las funciones de operaciones definidas para controlar el led.

```
static struct file_operations led_simple_ops = {
        .owner          = THIS_MODULE,
        .write          = led_write,
        .open           = led_open,
        .release        = led_close,
};
```

4.2.1 Control de GPIO en STM32MP157

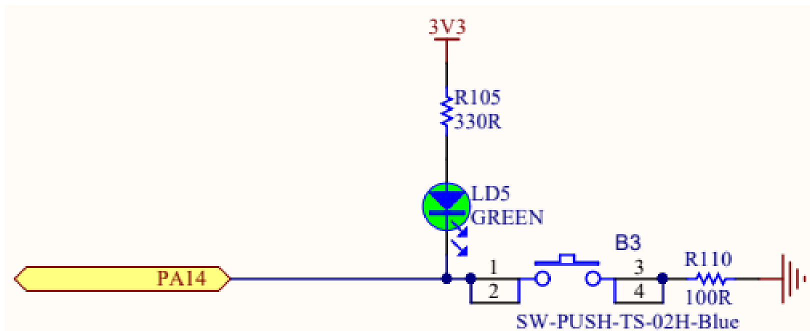

Figura 8. Esquemático del LED LD5 [9]

Como hemos visto anteriormente, el led LD5 que vamos a operar está conectado al pin PA14 y se activa a nivel bajo. Es decir, para encender este led solamente necesitamos configurar el pin PA14 para que emita un “0”.

El pin PA14 es un pin GPIO del grupo A, numero 14. GPIO (*General Purpose I/O Ports*) significa puertos de entrada/salida de propósito general, sirve para realizar controles simples a dispositivos. Por una parte, cuando el puerto está configurado a modo de entrada, podemos conectar el IO a un botón o sensor externo para recibir las señales entrantes externo. Por otra parte, cuando se utiliza como salida, podemos controlar el funcionamiento de los dispositivos externos emitiendo señales de nivel alto o bajo.

El control de los LED implica el uso de múltiples registros de control, incluido el reloj RCC y algunos registros de control GPIO. En la manual de referencia de STM32MP157 podemos encontrar todos los registros de configuración para GPIO. A continuación, vamos a ver los detalles de los registros mínimos necesarios para encender un LED.

4.2.1.1 Configuración del reloj

Según la página P522 de la manual de referencia de STM32MP157, podemos deducir que el reloj del GPIO se controla mediante el registro PLL4 del RCC.

10.7.43 RCC PLL4 Control Register (RCC_PLL4CR)

Address offset: 0x894

Reset value: 0x0000 0000

This register is used to control the PLL4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DIVREN	DIVQEN	DIVPEN	Res.	SSCG_CTRL	PLL4RDY	PLLON
									rw	rw	rw		rw	r	rw

Bit 1 **PLL4RDY**: PLL4 clock ready flag
Set by hardware to indicate that the PLL4 is locked.
0: PLL4 unlocked (default after reset)
1: PLL4 locked

Bit 0 **PLLON**: PLL4 enable
Set and cleared by software to enable the PLL4.
Cleared by hardware when entering Stop, LP-Stop, LPLV-Stop, or Standby mode.
Note that DIVPEN, DIVQEN and DIVREN of PLL4 must be set to '0' before setting PLLON to '0'. refer to *Section : PLL disabling procedure* for details.
0: PLL4 OFF (default after reset)
1: PLL4 ON, and ref4_ck is provided to the PLL4

CSDN @河边小乌龟爬

Figura 9. RCC PLL4 registro de control [10]

Por eso, necesitamos habilitar el bit 0 de este registro y esperar a que esté bloqueado cuando intentamos utilizar los dispositivos, es decir, en la función *open()*.

Debido a que el STM32MP157 consta de dos núcleos, cortex-m4 y cortex-A7, y los GPIO también puede ser controlado por estos dos núcleos respectivamente. Necesitamos habilitar el bit 0 del registro RCC_MP_AHB4ENSETR para que el reloj sea válido para GPIOA en el núcleo A7, que es más apropiado para nuestro desarrollo.

10.7.155 RCC AHB4 peripheral enable for MPU set register (RCC_MP_AHB4ENSETR)

Address offset: 0xA28

Reset value: 0x0000 0000

This register is used to set the peripheral clock enable bit of the corresponding peripheral to '1'. It shall be used to allocate a peripheral to the MPU. Writing '0' has no effect, reading will return the effective values of the corresponding bits. Writing a '1' sets the corresponding bit to '1'.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
					rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Figura 10. Habilitación de periféricos RCC AHB4 para registro de MPU [10]

4.2.1.2 Configuración del modo de GPIO

GPIOx_MODER se utiliza para configurar el modo de GPIO, dispone de cuatro modos: entrada, salida general, funciones alternativas y analógico. Este registro consta de 32 bits, involucrando 16 GPIO, y la configuración de cada GPIO emplea dos bits. Para conseguir que PA14 emita una señal de nivel bajo, necesitamos configurar los bits 28 y 29 de este registro a 01 (modo de salida), y si queremos leer el estado del led, los configuramos a 00 (modo de entrada).

13.4.1 GPIO port mode register (GPIOx_MODER) (x = A to K, Z)

Address offset:0x00

Reset value: 0xFFFF FFFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MODER[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figura 11. Registro de modo del puerto GPIO [10]

En consecuencia, la función *open()* en nuestra estructura *file_operations* está definida de esa manera:

```
static int led_open(struct inode *inode, struct file *filp) {
        // Enalbe PLL4
        *RCC_PLL4CR |= (1<<0);
        while ((*RCC_PLL4CR & (1<<1)) == 0);
        // Enable gpioA for A7 core
        *RCC_MP_AHB4ENSETR |= (1<<0);
        // Set PA14 to output mode
        *GPIOA_MODER &= ~(3<<28); // PA14, set bit[29:28] to 0x00
        *GPIOA_MODER |= (1<<28); // Set bit[29:28] to 01
        return 0;
}
```

4.2.1.3 Configuración de la salida

Hay dos métodos de configuración para la salida del puerto PA14. Por un lado, podemos escribir directamente en el registro GPIOx_ODR, que contiene el valor de salida de los puertos GPIO, con una longitud de 32 bits, donde los bits 0 a 16 representan los valores de salida correspondientes de GPIOx1 a GPIOx16. Por otro lado, podemos utilizar el registro GPIOx_BSRR para establecer y restablecer los diferentes puertos, sus bits 0 a 15 se utilizan para establecer los puertos GPIO correspondientes a "1" y los bits 16 a 31 sirven para configurarlos a "0".

13.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A to K, Z)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BR[15:0]:** Port x reset I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BS[15:0]:** Port x set I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

Figura 12. Registro para establecer/restablecer los puertos GPIO

Para resumir, para encender el LED vamos a configurar el bit 30 a 1, es decir, restablecer el puerto PA14 con el objetivo de emitir un 0 a la salida. Por el contrario, configuramos el bit 14 a 1 en el momento que queremos apagarlo. Por lo tanto, tenemos la función *write()* definida de siguiente forma mostrada:

```
static ssize_t led_write(struct file *filp, const char __user *buf,
                                size_t count, loff_t *ppos)
{
        char val;   // led status
        int ret;
        ret = copy_from_user(&val, buf, 1);
        if (ret < 0){
                printk(KERN_ERR "led_write: failed to copy %d\n", ret);
                return ret;
        }
        // led control
        if (val == LEDON){
                *GPIOA_BSRR = (1<<30);
        }
        else if (val == LEDOFF){
                *GPIOA_BSRR = (1<<14);
        }
        return 0;
}
```

Hay que tener en cuenta que estamos en el espacio de kernel y debemos utilizar la función *copy_from_user* para leer los datos que vienen desde la aplicación que está ejecutando en el espacio de usuario.

4.2.2 Gestión de memoria

4.2.2.1 Introducción de MMU

En una memoria, los datos están almacenos en unidades de bytes, cada unidad de byte se asigna una dirección de memoria única, también conocida como dirección física. Es muy peligroso acceder directamente a la memoria física en el entorno Linux, para evitar errores del sistema causados por modificación de los datos en la memoria incorrectamente, Linux introduce la Unidad de gestión de memoria (en inglés, *Memory management unit, MMU*). MMU tiene las siguientes funciones principales:

• Conversión entre la dirección virtual y la dirección física: la CPU puede ejecutarse en una memoria virtual, y en general, la memoria virtual es mucho más grande que la memoria física real, por lo que la CPU puede ejecutar aplicaciones que consumen mucha memoria.

• Protección de memoria: la tabla de paginación de la MMU contiene los permisos para leer, escribir y ejecutar algunos bloques de memoria específicos. La MMU verificará si la CPU se encuentra actualmente en modo privilegiado o en modo de usuario para confirmar si está permitido a operar este block de memoria para evitar que la memoria se modifique sin autorización.

En un sistema que emplea MMU, cuando el programa intenta controlar un determinado registro, la CPU enviará una dirección virtual a la MMU, y luego la MMU traducirá la dirección virtual en una dirección física real para la operación. Los diferentes programas en proceso tienen sus propios espacios de direcciones virtuales y para cada programa está prohibido modificar la dirección física utilizada en otro proceso, por lo que los procesos están aislados y no interfieren entre sí.

4.2.2.2 Mapeo de direcciones

Por lo tanto, al cargar el módulo del driver, necesitamos obtener las direcciones virtuales correspondientes a las direcciones físicas que será manejado y liberarlas cuando se descarga el driver. Utilizaremos las funciones *ioremap* y *iounmap*, sus definiciones se pueden encontrar en el archivo *arch/arm/include/asm/io.h*. Además, procesamos la carga y descarga del driver en los módulos *init* y *exit* respectivamente, los métodos específicos hablaremos en detalle en la sección 4.7.

Para conocer las direcciones físicas de los registros, consultamos el manual de referencia. Las direcciones base de RCC y GPIOA se encuentran en la página 162, y la dirección de cada registro se puede obtener sumando la dirección base y el desplazamiento descrito en su definición.

```
#define RCC_BASE_ADDR                (0x50000000)
#define GPIOA_BASE_ADDR              (0x50002000)
#define RCC_PLL4CR_ADDR              (RCC_BASE_ADDR + 0x894)
#define RCC_MP_AHB4ENSETR_ADDR       (RCC_BASE_ADDR + 0xA28)
#define GPIOA_MODER_ADDR             (GPIOA_BASE_ADDR + 0x00)
```

```
#define GPIOA_BSRR_ADDR                (GPIOA_BASE_ADDR + 0x18)

static volatile unsigned int *RCC_PLL4CR;
static volatile unsigned int *RCC_MP_AHB4ENSETR;
static volatile unsigned int *GPIOA_MODER;
static volatile unsigned int *GPIOA_BSRR;

static int __init led_init(void){
        // 0x50000000 + 0x894
        RCC_PLL4CR = ioremap(RCC_PLL4CR_ADDR, 4); // 4 bytes
        // 0x50000000 + 0xA28
        RCC_MP_AHB4ENSETR = ioremap(RCC_MP_AHB4ENSETR_ADDR, 4);
        // 0x50002000 + 0x00
        GPIOA_MODER = ioremap(GPIOA_MODER_ADDR, 4);
        // 0x50002000 + 0x18
        GPIOA_BSRR = ioremap(GPIOA_BSRR_ADDR, 4);

    ......
}

static void __exit led_exit(void){
        // led unmap
        led_unmap();

    ......
}

void led_unmap(void)
{
        iounmap(RCC_PLL4CR);
        iounmap(RCC_MP_AHB4ENSETR);
        iounmap(GPIOA_MODER);
        iounmap(GPIOA_BSRR);
}
```

4.3 Número de dispositivo

El número de dispositivo es equivalente a un identificador del dispositivo, consta de un número mayor y un número menor. [11] El número de dispositivo mayor está asociado directamente con el código del driver, en cuando el usuario intenta manipular un dispositivo, el kernel emplea este número para encontrar el driver propio que debe ejecutarse. El número menor identifica cada dispositivo que utiliza el mismo fichero de driver. Por ejemplo, con un solo driver de LED somos capaz de controlar todos los LED en la placa, en este caso tendrán el mismo número de dispositivo mayor, por lo tanto, utilizamos el número de dispositivo menor para distinguir estos LED diferentes.

4.3.1 Definición

Podemos encontrar la definición del número de dispositivo (dev_t) en el fichero *include/linux/type.h* en el kernel de Linux:

```
typedef u32 __kernel_dev_t;
typedef __kernel_dev_t          dev_t;
```

Así como, el fichero *include/linux/kdev_t.h* proporciona algunas funciones de operación del número de dispositivo:

```
#define MINORBITS   20
#define MINORMASK ((1U << MINORBITS) - 1)

#define MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)        (((ma) << MINORBITS) | (mi))
```

Podemos deducir que dev_t es un tipo de datos de 32 bits, en el que los 12 bits superiores son el número de dispositivo mayor y los 20 bits inferiores son el número de dispositivo menor.

4.3.2 Configuración

La configuración del número de dispositivo se lleva a cabo en el módulo *init*, y hay dos formas de hacerlo. Podemos utilizar el comando `cat/proc/devices` para comprobar los números de dispositivo que están de uso en el sistema, y asignar al dispositivo que vamos a utilizar un número disponible. Sin embargo, esta forma es muy engorrosa y puede conllevar conflictos. Será mejor que usemos el otro método: Solicitar un número de dispositivo directamente al sistema con la función *alloc_chrdev_region* que está definida en */linux/fs/char_dev.c*, y los números mayor y menor se conseguirán simplemente empleando las funciones mencionadas anteriormente *MAJOR(dev)* y *MINOR(dev)*. Y si ya tenemos el número de dispositivo, solo necesitamos registrarlo mediante la función *register_chrdev_region*.

```
#define LED_COUNT  1

// device number
```

```
	if (my_led.major) {	// if major is defined
		my_led.dev_nb = MKDEV(my_led.major, 0);
		ret = register_chrdev_region(my_led.dev_nb, LED_COUNT, LED_NAME);
		if(ret < 0) {
			pr_err("register_chrdev_region failed on %s, ret=%d\r\n", LED_NAME, ret);
			led_unmap();
		}
	} else {
		ret = alloc_chrdev_region(&my_led.dev_nb, 0, LED_COUNT, LED_NAME);
		if(ret < 0) {
			pr_err("alloc_chrdev_region failed on %s, ret=%d\r\n", LED_NAME, ret);
			led_unmap();
		}
		my_led.major = MAJOR(my_led.dev_nb);
		my_led.minor = MINOR(my_led.dev_nb);
	}
	printk("my_led: major=%d, minor=%d\r\n", my_led.major, my_led.minor);
```

Al final en el módulo exit, necesitamos liberar el número de dispositivo cuando ya no utilizamos este dispositivo:

```
unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
```

4.4 Registro del dispositivo

Como hemos hablado en la instrucción, para agregar el dispositivo que queremos manejar necesitamos registrar un nuevo objeto *cdev* al kernel. El proceso es: definir un objeto *cdev*, iniciarlo mediante la función *cdev_init* y añadirlo al kernel con la función *cdev_add*. Las definiciones de estas funciones podemos encontrar en *\linux\fs\char_dev.c*.

```
	my_led.cdev.owner = THIS_MODULE;
	cdev_init(&my_led.cdev, &led_simple_ops);
	ret = cdev_add(&my_led.cdev, my_led.dev_nb, LED_COUNT);
	if(ret < 0){
		unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
	}
```

Podemos observar que, para iniciar cdev, hacemos uso de la colección de funciones de operación de archivos *led_simple_drv* que hemos definido en pasos anteriores, y añadimos este dispositivo al kernel con la ayuda del número de dispositivo *dev_nb* que hemos solicitado en la sección 4.3.2.

Así mismo, tenemos que eliminar el dispositivo cuando descargamos el driver en el módulo *exit* con la función *cdev_del*.

```
        cdev_del(&my_led.cdev);
```

4.5 Nodo de dispositivo

Una vez tenemos el dispositivo registrado, es imprescindible crear un fichero correspondiente en la carpeta */dev/* para que los controles de la aplicación puedan efectuarse sobre el dispositivo tras operando este fichero relacionado, que denominamos nodo de dispositivo.

En nuestro sistema Linux embebido, se aprovecha la herramienta *mdev* para llevar a cabo el control automático de los nodos de dispositivos. *mdev* es una aplicación de usuario que puede detectar los estados de los dispositivos en el sistema, añadirá los ficheros de dispositivos que faltan cuando cargamos el driver y eliminará los que ya no están de uso cuando descargamos el driver.

En este apartado vamos a ver cómo crear y eliminar automáticamente los nodos de archivos de dispositivos a través de *mdev*.

4.5.1 Creación y eliminación de clase

El proceso de la creación automática de un nodo de dispositivo se realiza en la función *init*. En primer lugar, se exige crear una clase *class*. *class* es una estructura que está definida en *include/Linux/device.h*. *class_create* es la función para crear esta clase, recibe 2 argumentos como entrada. *owner*, que generalmente es *THIS_MODULE*, y *name*, que es el nombre de la clase. Como resultado, devuelve el puntero a la estructura *class* creado.

```
#define LED_CLASS_NAME   "myled_class"


        my_led.class = class_create(THIS_MODULE, LED_CLASS_NAME);
        if (IS_ERR(my_led.class)) {
                cdev_del(&my_led.cdev);
                return -1;
        }
```

Igualmente, es esencial eliminar esta clase cuando descargamos el driver. La función para procesar la eliminación es *class_destroy*.

```
        class_destroy(my_led.class);
```

4.5.2 Creación y eliminación de dispositivo

No podemos obtener el nodo de dispositivo creado con la clase sola. Se requiere además definir el dispositivo en esta clase con la función *device_create*. Así mismo, hay que destruirlo en la función *exit*.

```
#define LED_NAME    "myled"
```

```
    my_led.device = device_create(my_led.class, NULL, my_led.dev_nb, NULL, LED_NAME); //dev/myled
    if (IS_ERR(my_led.device)) {
        class_destroy(my_led.class);
        return -1;
    }
```

```
    device_destroy(my_led.class, my_led.dev_nb);
```

Por consiguiente, cuando registramos el LED en el kernel, el fichero */dev/myled*, es decir el nodo de dispositivo correspondiente al dispositivo, se creará automáticamente, y cuando descargamos el driver se eliminará automáticamente.

4.6 Archivos de cabecera y licencias

Para completar, añadimos los archivos de cabecera necesarios y la licencia.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/mutex.h>
#include <linux/wait.h>
#include <linux/uaccess.h>
#include <asm/io.h>
#include <linux/device.h>

MODULE_LICENSE("GPL");
```

4.7 Carga y descarga del driver

Una vez se dispone el driver desarrollado podemos compilarlo utilizando un compilador cruzado a un módulo con extensión *.ko,* y ejecutar el comando `insmod driver_name.ko` para cargar el driver. Así mismo, ejecutar el comando `rmmod driver_name.ko` para descargarlo. Las funciones invocadas son *led_init* y *led_exit* que ya tenemos completado:

```
module_init(led_init);
module_exit(led_exit);
```

Para resumir, en el módulo *led_init*, realizamos las siguientes acciones:

1. Conversión de las direcciones físicas que vamos a manejar a las direcciones virtuales para que esté disponible por la aplicación.
2. Obtención y registro del número de dispositivo.
3. Iniciar y añadir *cdev* al sistema, empleando el número de dispositivo y la estructura *file_operations* definida específicamente.
4. Crear la clase y el dispositivo para realizar la creación y eliminación del archivo /dev/device_name (nodo de dispositivo) automáticamente.

En cuanto al módulo *led_exit*, tiene las siguientes funcionalidades:

1. Liberación de las direcciones virtuales.
2. Eliminar el dispositivo y la clase (nodo de dispositivo).
3. Eliminar el dispositivo *cdev* desde el sistema.
4. Liberar el número de dispositivo.

4.8 Aplicación de verificación y Makefile

Con el objetivo de testear el driver que hemos implementado, creamos una aplicación Linux sencilla *testdrv.c* que enciende y apaga el LED, tiene principalmente las siguientes funciones:

- Verificación del número de argumentos de entrada y proponer métodos de ejecutar.
- Lectura del valor input de usuario.
- Abrir el fichero, escribirlo y cerrarlo, esto indica, realizar controles en el nodo de dispositivo con las funciones *open()*, *write()* y *close()* realizada en *file_operations*.

Además, para indicar en qué dispositivo queremos operar, se lanza con el nodo de dispositivo */dev/myled* que creamos para el led LD5 como argumento.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
/* Usage:
ledtest /dev/myled on
ledtest /dev/myled off
*/
int main(int argc, char **argv)
{
        int fd, ret;
        char status = 0;
        if (argc != 3){
```

```
		printf("Usage: %s <dev> <on|off>\n", argv[0]);
		printf("  eg: %s /dev/myled on\n", argv[0]);
		printf("  eg: %s /dev/myled off\n", argv[0]);
		return -1;
	}
	// open
	fd = open(argv[1], O_RDWR);
	if (fd < 0){
		printf("File not found: %s\n", argv[1]);
		return -1;
	}
	// write
	if (strcmp(argv[2], "on") == 0){ // led on
		status  = 1;
	}
	ret = write(fd, &status, 1);
	if(ret < 0){
		printf("Failed to control LED.\r\n");
		close(fd);
		return -1;
	}
	ret = close(fd);
	if(ret < 0){
		printf("Failed to close file.\r\n");
		return -1;
	}
	return 0;
}
```

Para la validación del driver, otro archivo importante es el Makefile. Este fichero tiene el siguiente contenido:

- Reglas de compilación. Se compila los programas con la ayuda del código fuente de kernel de Linux utilizando el compilador cruzado configurado, para obtener como objetivo el driver *led_drv*, y tambien se obtiene la aplicación *ledtest*.
- La función *clean* para eliminar los ficheros compilados.
- La función *deploy* para transmitir el programa de driver y la aplicación compiladas a la placa utilizando su dirección IP mediante el comando scp que hemos presentado en la sección 2.2.

```
KERN_DIR = <KERNEL PATH>

obj-m   += led_drv.o
all:
        make -C $(KERN_DIR) M=`pwd` modules
        $(CROSS_COMPILE)gcc -o ledtest ledtest.c
clean:
        make -C $(KERN_DIR) M=`pwd` modules clean
        rm -rf modules.order
        rm -f ledtest
deploy:
        scp *.ko root@<board ip address>:./led_simple    // ip if the board
        scp ledtest root@<board ip address>:./led_simple
```

4.9 Resultados

Después de compilar, obtendremos el archivo de driver con extensión *.ko* y archivos de prueba con extensión *.c*, y podemos utilizar el comando make deploy que mencionamos antes para pasar los archivos al directorio */led_simple* de la placa.

```
cai@cai-virtual-machine:~/cai_tfg/led_simple$ make deploy
scp *.ko root@192.168.18.191:./led_simple
led_drv.ko                              100%  165KB 148.7KB/s   00:01
scp ledtest root@192.168.18.191:./led_simple
ledtest                                 100%   14KB 132.5KB/s   00:00
```

Figura 13

Una vez tenemos los programas disponibles, podemos cargar el driver a través del comando insmod led_drv.ko, con lo cual se ejecuta el módulo init del driver. Se puede observar que el registro ha tenido éxito y ha devuelto los números de dispositivo mayor y menor.

```
root@stm32mp1:~/led_simple# insmod led_drv.ko
[ 4723.803213] led_drv: loading out-of-tree module taints kernel.
[ 4723.808715] /home/cai/cai_tfg/led_simple/led_drv.c led_init 120
[ 4723.818793] my_led: major=242, minor=0
```

Figura 14

Con el comando lsmod se muestra la lista de drivers en el sistema.

```
root@stm32mp1:~/led_simple# lsmod
Module                  Size  Used by
led_drv                16384  0
```

Figura 15

Además, podemos comprobar que el nodo de dispositivo */dev/myled* se ha creado automáticamente cuando se registra el driver, y el número de dispositivo de *myled* coincide con lo que ha devuelto anteriormente con el comando cat /proc/devices.

```
root@stm32mp1:~/led_simple# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
226 drm
242 myled
243 rpmb
```

Figura 16

Después de comprobar que todo está correcto, podemos ejecutar la aplicación de verificación con los comandos definidos. El LED LD5 se enciende con ./ledtest /dev/myled_on y se apaga mediante ./ledtest /dev/myled_off.

```
root@stm32mp1:~/led_simple# ./ledtest
Usage: ./ledtest <dev> <on|off>
  eg: ./ledtest /dev/myled on
  eg: ./ledtest /dev/myled off
```

Figura 17

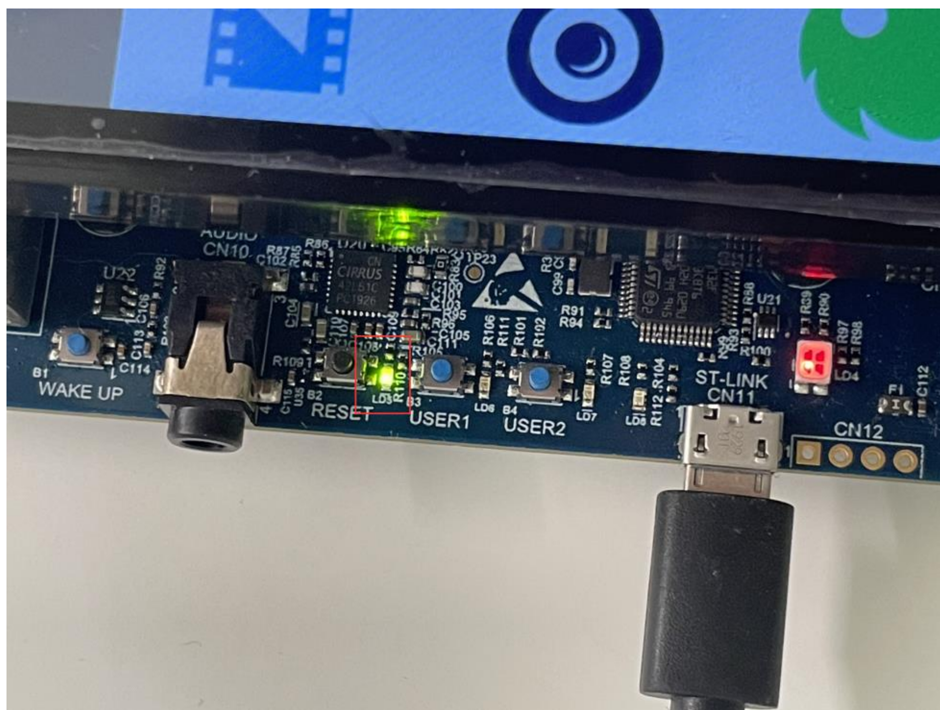

Figura 18

Se puede observar que el LED LD5 está encendido correctamente.

Cuando finaliza la validación, se recomienda descargar el driver con el comando *rmmod*. Se ejecutará el módulo *exit* del programa que realiza la liberación del número de dispositivo, eliminación del nodo de dispositivo, etc.

```
root@stm32mp1:~/led_simple# rmmod led_drv.ko
```

Figura 19

Capítulo 5. Implementación de dispositivos y drivers de plataforma

En el programa de driver del dispositivo de caracteres anterior, siempre que llamemos a la función *open()* para abrir el archivo del dispositivo correspondiente, podemos usar la función *read()/write()* para controlar el hardware a través de la interfaz de operación de archivos *file_operations*. Esta forma de desarrollo es muy simple e intuitiva, pero no es óptimo desde el punto de vista del diseño de software. Tiene el problema de que la información del dispositivo y los códigos del controlador se mezclan en un solo fichero, por lo que una vez que la información del hardware cambia, se debe modificar todo el código fuente del driver.

Con el fin de mejorar la mantenibilidad y reutilización del código, en la versión 2.6 de kernel Linux propone un modelo de bus, dispositivo y driver. [12]

5.1 Modelo de bus - driver - dispositivo

En este modelo los códigos que escribimos se dividen en dos partes: dispositivo y driver. El dispositivo es responsable de proporcionar informaciones de hardware y el código de driver utiliza estos recursos de dispositivos para hacer operaciones. Las dos partes se conectan mediante el Bus, y la relación entre los tres se muestra en la siguiente figura:

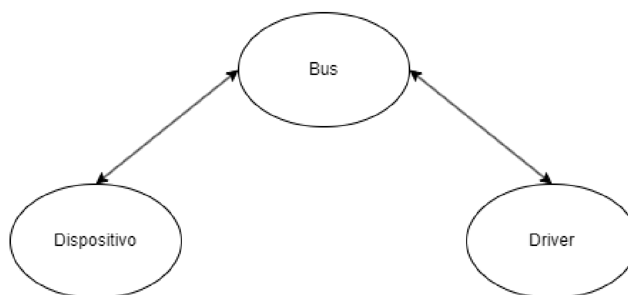

Figura 20. Modelo bus - driver - dispositivo

En kernel de Linux, se utiliza la estructura *bus_type* del archivo */include/linux/device.h* para representar el bus. Los dispositivos y drivers se definen por las estructuras *device* y *device_driver* del mismo fichero. [13]

En el bus se gestionan dos listas enlazadas, una de dispositivos y otra de drivers. Cuando registramos un nuevo driver en el sistema, se insertará en la lista enlazada de gestión del driver. De forma similar, cuando registramos un nuevo dispositivo en el sistema, será insertado en la lista de administración de dispositivos.

Cada vez que se inserta un elemento en cualquiera de las dos listas, el bus realizará un método de *match* que viene de la estructura *bus_type* para buscar el driver/dispositivo que soporta este dispositivo/driver respectivamente. (Hay varias formas de hacer emparejamiento entre ellos, la más fácil es comprobar si tienen el mismo nombre). Cuando coinciden, se invocará la función *probe* de la estructura *device_driver* del driver, con la que podemos obtener los recursos definidos en la estructura *device* para hacer operaciones personalizadas. Y cuando el dispositivo o el controlador se elimina, se llamará al método *remove* de la estructura *device_driver*. [13] Las funciones *probe* y

remove que hemos mencionado anteriormente se deben elaborar por nosotros cumpliendo nuestros propios requisitos.

5.2 Introducción a dispositivos y drivers de plataforma

Sin embargo, el concepto de bus en el modelo de driver es una abstracción a nivel de software, no es equivalente al concepto de bus físico que sirve como una línea de comunicación común para transmitir información entre el chip y varios periféricos en nuestro SoC (*System on a chip*).

En general, para los tipos comunes de buses físicos como I2C, SPI y USB, el kernel de Linux creará automáticamente el bus de driver correspondiente, por lo que los dispositivos I2C, los dispositivos SPI y los dispositivos USB se registran y se montan en el bus correspondiente. No obstante, hay muchos dispositivos con estructuras simples que no tienen un bus físico correspondiente, como led, zumbador, botón, etc.

Para que el desarrollo del driver de este tipo de dispositivo que no tienen un bus físico correspondiente también siga el modelo del bus – driver – dispositivo, el kernel de Linux introduce un bus virtual, el bus de la plataforma. Estos dispositivos se denominan dispositivos de plataforma y los drivers de dispositivos correspondientes se denominan drivers de plataforma.

La idea central de driver y dispositivo de plataforma sigue siendo el modelo de driver y dispositivo. Los dispositivos y drivers de plataforma están representados por las estructuras de *platform_device* y *platform_driver* correspondiente, que heredan las estructuras *device* y *device_driver*.

5.3 Dispositivo de plataforma

Creamos un fichero *led_device.c* para describir y registrar nuestro dispositivo de plataforma, el LED LD5. Se define como una instancia de la estructura *platform_device* de la siguiente forma:

```
static struct platform_device led_device = {
        .name = "my_led",
        .dev = {
            .release = led_dev_release,
         },
        .num_resources = ARRAY_SIZE(led_resources),      // number of resources
        .resource = led_resources,                    // resource
};
```

La definición completa de la estructura *platform_device* se describe en el archivo */include/linux/platform_device.h*. En nuestro *led_device*, hay que destacar los siguientes miembros:

- *name*: Nombre del dispositivo. Cuando el bus se empareja utilizando la función *match*, comparará si los nombres del dispositivo y el driver son mismos.

- *dev*: Abstracción de la estructura *device*.
- *num_resources*: Numero de recursos.
- *resources*: Recursos proporcionados por el dispositivo de plataforma, es decir, las informaciones del LED. En el driver correspondiente se obtienen estos recursos para hacer operaciones sobre el dispositivo.

5.3.1 Definición de recursos

En nuestro programa, los recursos requeridos son principalmente la información de los registros que operaremos, que son: los registros para habilitar reloj en Core A7 para GPIOA, el registro para configurar el modo de salida del PA14, y el que controla la salida. Utilizando las direcciones que ya estaban definidos en la sección 4.2.2.2 Mapeo de direcciones, diseñamos la estructura *led_resources* de la siguiente forma:

```
#define REGISTER_LENGTH                 4

static struct resource led_resources[] = {
    [0] = {
            .start  = RCC_PLL4CR_ADDR,
            .end    = (RCC_PLL4CR_ADDR + REGISTER_LENGTH - 1),
            .flags  = IORESOURCE_MEM,
      },
      [1] = {
            .start  = RCC_MP_AHB4ENSETR_ADDR,
            .end    = (RCC_MP_AHB4ENSETR_ADDR + REGISTER_LENGTH - 1),
            .flags  = IORESOURCE_MEM,
      },
      [2] = {
            .start  = GPIOA_MODER_ADDR,
            .end    = (GPIOA_MODER_ADDR + REGISTER_LENGTH - 1),
            .flags  = IORESOURCE_MEM,
      },
      [3] = {
            .start  = GPIOA_BSRR_ADDR,
            .end    = (GPIOA_BSRR_ADDR + REGISTER_LENGTH - 1),
            .flags  = IORESOURCE_MEM,
      },
};
```

Los miembros *start* y *end* significan las direcciones inicial y final del recurso, *flag* se aplica para indicar el tipo de recurso. Los recursos incluyen I/O, DMA, bus y otros tipos,

en *include/linux/ioport.h* están definidos todos los tipos opcionales. Utilizamos *IORESOURCE_MEM* para representar recursos de memoria.

5.3.2 Registro de un dispositivo de plataforma

Después de definir e inicializar la estructura de *platform_device*, debemos registrarla en el bus de plataforma empleando la función *platform_device_register()*, lo realizamos en el módulo *init*.

```
static int __init led_dev_init(void) {
    return platform_device_register(&led_device);
}

module_init(led_dev_init);
```

Tenemos que eliminar este dispositivo en el módulo *exit* igualmente.

```
static void __exit led_dev_exit(void) {
    platform_device_unregister(&led_device);
}

module_exit(led_dev_exit);
```

5.4 Driver de plataforma

En *led_driver.c*, realizamos las mismas funciones que el driver desarrollado en el Capítulo 4, como la definición de funciones de operación de archivos, carga y descarga de dispositivos, etc. La única diferencia es que la información del hardware ya no necesita ser descrita en este programa, sino que se obtiene del dispositivo de plataforma definido en *led_device.c*, pero es esencial definir el driver de plataforma correspondiente:

```
static struct platform_driver led_driver = {
        .driver         = {
                .name   = "my_led",
        },
        .probe          = led_probe,
        .remove                 = led_remove,
};
```

- *driver*: Como hemos mencionado anteriormente, tenemos que asignar al driver el mismo nombre que el dispositivo para que coincidan. En la estructura *platform_driver*, el nombre se define en el miembro *driver*, que es una abstracción de la estructura *device_driver*. [14]
- *probe*: Puntero de función. Cuando hemos conseguido el emparejamiento del dispositivo y driver se llamará a esta función. Generalmente en esta función

realizamos las operaciones que hicimos en el módulo *init* del driver anterior, como añadir *cdev* al sistema, crear el nodo de dispositivo, etc.

- *remove*: Se puede decir que la función *led_remove* es equivalente al módulo *exit* que implementamos antes, sirve para liberar el número de dispositivo, descargar el driver, etc.

5.4.1 Obtención de recursos

Aparte de las inicializaciones del dispositivo, otra función importante de *probe* es obtener los recursos definidos en el dispositivo de plataforma. Para hacer esto, se utiliza la función *platform_get_resource* que devuelve un puntero a la estructura *resource* que hemos definido. En nuestro caso, accedemos al atributo *start* para obtener las direcciones de los registros que vamos a manipular, y ya podemos seguir los mismos pasos descritos en 4.2.1 para controlar el LED operando *led_pins[i]*.

```
        static int led_pins[4];
struct resource *led_resource;

// Get the resource
        for (i = 0; i < 4; i++) {
    led_resource = platform_get_resource(dev, IORESOURCE_MEM, i);
        if (!led_resource) {
            printk("Failed to get resource.\r\n");
            return -1;
        }
    led_pins[i] = led_resource->start;
        }
```

5.4.2 Registro de un driver de plataforma

De igual modo, el driver de plataforma se debe registrar en el sistema en el módulo *init* y eliminar en *exit* cuando no se utiliza.

```
static int __init led_init(void) {
  return platform_driver_register(&led_driver);
}

static void __exit led_exit(void) {
  platform_driver_unregister(&led_driver);
}

module_init(led_init);
module_exit(led_exit);
```

5.5 Resultados

En cuanto a la aplicación de verificación, sigue siendo lo mismo que en el apartado 4.8 ya que realmente los cambios de implementación no afectan al control del led de usuario.

No obstante, tenemos que modificar el objetivo del Makefile para generar un archivo *.ko* para el dispositivo de plataforma y otro para el driver de plataforma. También, es necesario cargar los dos ficheros ejecutando el comando insmod.

```
obj-m += led_device.o led_driver.o
```

El log destaca que el emparejamiento del dispositivo y driver de plataforma ha tenido éxito.

```
root@stm32mp1:~/led_platform# insmod led_driver.ko
[  635.456259] Match successfully.
[  635.458293] my_led: major=242, minor=0
```

Figura 21

Una vez se disponen el driver de plataforma y el dispositivo de plataforma cargados en la placa, podemos comprobar que estén correctamente añadidos al sistema de las siguientes formas:

```
ls /sys/bus/platform/devices
```

```
root@stm32mp1:~/led_platform# insmod led_device.ko
[  449.522897] led_device: loading out-of-tree module taints kernel.
root@stm32mp1:~/led_platform# ls /sys/bus/platform/devices
10000000.m4                 49000000.usb-otg                58000000.dma-controller              5c00a000.tamp               reg-dummy
10000000.sram               4c000000.hwspinlock             58009000.crc                         5c00a000.tamp:reboot-mode   regulatory.0
2ffff000.sram               4c001000.mailbox                5800a000.ethernet                    Fixed MDIO bus.0            snd-soc-dummy
4000b000.audio-controller   50000000.rcc                    5800d000.usbh-ehci                   ahb                         soc
4000e000.serial             50001000.pwr                    59000000.gpu                         alarmtimer.0.auto           soc:pin-controller-z@54004000
40010000.serial             50001014.pwr_mcu                5a000000.dsi                         arm-pmu                     soc:pin-controller@50002000
40012000.i2c                50001020.pwr                    5a001000.display-controller          cpufreq-dt                  sound
40016000.cec                5000d000.interrupt-controller   5a002000.watchdog                    dfd00000.framebuffer        stm32-cpufreq
4400b000.sai                50020000.syscon                 5a006000.usbphyc                     firmware:scmi0              timer
4400b004.audio-controller   50025000.vrefbuf                5c002000.i2c                         hdmi-audio-codec.1.auto     vin
4400b024.audio-controller   50028000.thermal                5c002000.i2c:stpmic@33:onkey         led                         wifi-pwrseq
48000000.dma-controller     54001000.cryp                   5c002000.i2c:stpmic@33:regulators    my_led.0
48001000.dma-controller     54002000.hash                   5c004000.rtc                         pm_domain
48002000.dma-router         54003000.rng                    5c005000.efuse                       psci
```

Figura 22. Verificación del dispositivo de plataforma

```
ls /sys/bus/platform/drivers
```

```
root@stm32mp1:~/led_platform# ls /sys/bus/platform/drivers
Xilinx Watchdog          brcm-kona-reset      i2c-mux-pinctrl       palmas-pmic            simple-framebuffer     stm32-pwr-regulator   syscon-reboot-mode
act8945a-charger         brcmstb_nand         iio_hwmon             palmas-rtc             simple-pm-bus          stm32-qspi            tegra-udc
act8945a-regulator       cdns-spi             iio_sd_adc_mod        panel-simple           simple-reset           stm32-rng             tps65090-charger
ahci                     chipidea-usb2        imx_usb               physmap-flash          smsc911x               stm32-romem           tps65090-pmic
alarmtimer               ci_hdrc              iproc_nand            pinctrl-single         snd-soc-dummy          stm32-rproc           tps65217-pmic
arm-scmi                 cpcap-regulator      isp1760               pm8xxx-core            snps-udc-plat          stm32-usart           tps65218-pmic
arm-smc-mbox             cpufreq-dt           iwdg                  poweroff-gpio          soc-audio              stm32-usbphyc         tps6586x-gpio
armv7-pmu                denali-nand-dt       leds-gpio             pwm-backlight          sram                   stm32-vrefbuf         tps6586x-regulator
as3711-backlight         dw_dmac              leds_pwm              pwm-regulator          ssbi                   stm32_cpuidle         tps6586x-rtc
as3711-regulator         dw_mmc               m10v-ccu              pwrseq_emmc            st,stm32-i2s           stm32_exti            tps65910-gpio
as3722-pinctrl           dw_wdt               m_can_platform        pwrseq_simple          st,stm32-sai           stm32_fmc2_ebi        tps65910-pmic
as3722-power-off         dwc-eth-dwmac        macb                  ramoops                st,stm32-sai-sub       stm32_fmc2_nfc        tps65910-rtc
as3722-regulator         dwc2                 max77686-clk          rcar-lvds              st_thermal_mmap        stm32_hwspinlock      twl4030_gpio
as3722-rtc               dwmmc_exynos         max77686-pmic         reg-dummy              stm32-cec              stm32_pwr             twl4030_power
asoc-audio-graph-card    ehci-platform        max77686-rtc          reg-fixed-voltage      stm32-cpufreq          stm32_rtc             twl4030_reg
aspeed-clk               fsl-edma             max77802-pmic         restart-gpio           stm32-crc32            stm32f7-i2c           twl6030_reg
aspeed-g6-pinctrl        gpio-backlight       max8907-regulator     rfkill_gpio            stm32-cryp             stm32mp157-pinctrl    twl_rtc
aspeed_wdt               gpio-clk             max8907-rtc           rk808-regulator        stm32-display          stm32mp1_rcc          usb-conn-gpio
ast2600-clk              gpio-dwapb           mlb-usio-uart         rn5t618-regulator      stm32-display-dsi      stm_thermal           usb3503
atmel_flexcom            gpio-keys            msm_hsusb             rn5t618-wdt            stm32-dma              stmfx-pinctrl         usbmisc_imx
axp20x-regulator         gpio-regulator       my_led                rproc-srm-core         stm32-dmamux           stmmaceth             vexpress-regulator
basic-mmio-gpio          gpio-syscon          of_fixed_clk          rproc-srm-dev          stm32-dwmac            stmpe-ts              vexpress-reset
bcm-keypad               gpio-vbus            of_fixed_factor_clk   s2mps11-pmic           stm32-hash             stpmic1-regulator     vexpress-syscfg
bcm590xx-vregs           gpio-xilinx          ohci-platform         s5m8767-pmic           stm32-ipcc             stpmic1-wdt           vexpress-sysreg
bcm63138_nand            hci_bcm              optee                 sbsa-uart              stm32-lptimer          stpmic1_onkey         virtio-mmio
bcm6368_nand             hdmi-audio-codec     orion-mdio            sdhci-arasan           stm32-lptimer-timer    syscon                xadc
bcma-host-soc            hisi-gmac            palmas-gpio           sdhci-at91             stm32-mdma             syscon-poweroff       xhci-hcd
bdc                      i2c-demux-pinctrl    palmas-pinctrl        sdhci-omap             stm32-pm-domain        syscon-reboot         xilinx_spi
```

Figura 23. Verificación del driver de plataforma

El nodo de dispositivo */dev/myled_platform* se ha creado correctamente y se le asigna un numero de dispositivo 242.

```
root@stm32mp1:~/led_platform# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
226 drm
242 myled_platform
243 rpmb
```

Figura 24

De la misma forma, se ejecuta el programa, descarga tanto el dispositivo como el driver cuando completamos la verificación.

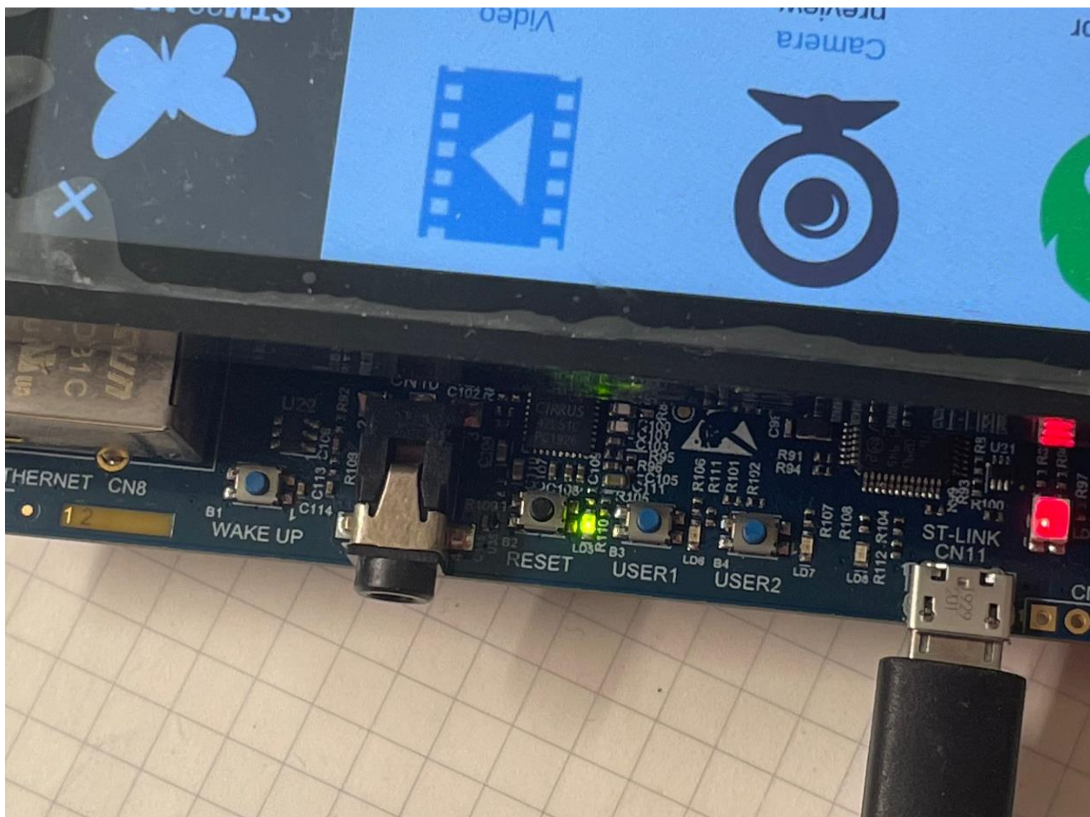

Figura 25. Resultado

```
root@stm32mp1:~/led_platform# rmmod led_driver.ko
root@stm32mp1:~/led_platform# rmmod led_device.ko
[ 1887.190016] Led device released.
```

Figura 26

Capítulo 6. Árbol de dispositivos

Sin embargo, todavía hay algunos problemas en el desarrollo con el modelo de dispositivo-plataforma-driver. Tomando nuestro programa LED como ejemplo, utilizamos LED LD5 que es un pin perteneciente al grupo GPIOA. Pero si queremos encender el LED LD7 que pertenece al grupo GPIOH, hay que utilizar otro archivo nuevo muy similar para definir este dispositivo de plataforma y los recursos relacionados. Por lo tanto, con la popularidad de los chips ARM, para describir miles de dispositivos, hay muchos archivos similares con códigos duplicados en el kernel de Linux, lo que hace que el kernel esté particularmente inflado. Con el propósito de resolver este problema, desde la versión 3.x Linux introduce el uso del árbol de dispositivos.

6.1 Introducción

El archivo que describe el árbol de dispositivos se llama DTS (*Device Tree Source*). La función principal de este archivo DTS es usar una estructura de árbol para describir los dispositivos a nivel de placa, es decir, la información de los periféricos en la placa de desarrollo, como la cantidad de CPU, la dirección de memoria base, qué dispositivos están conectados a la Interfaz SPI, etc. El tronco del árbol es el bus del sistema, y el controlador GPIO, el controlador I2C, el controlador SPI, etc. son ramas conectadas al bus del sistema. [15]

Los archivos de árbol de dispositivos DTS se compilarán en archivos binarios DTB, al arrancar, el kernel analizará los archivos DTB para obtener todas las informaciones de los periféricos. De esta forma, no necesitamos almacenar tantos archivos *.c* en el kernel para describir los dispositivos.

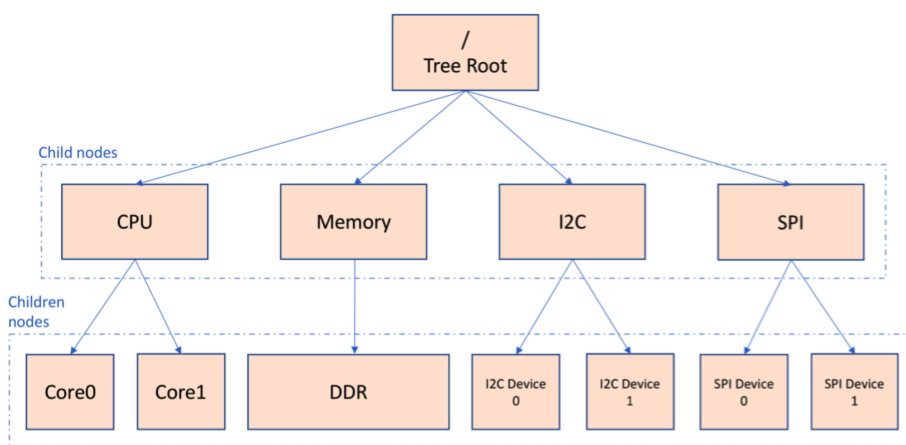

Figura 27. Estructura simple de un árbol de dispositivo [16]

6.2 Sintaxis del árbol de dispositivos

Para explicar la sintaxis del árbol de dispositivos, tomamos un trozo de código del stm32mp157f-dk2.dts del *\linux\arch\arm\boot\dts\stm32mp157f-dk2.dts* como ejemplo.

```
#include "stm32mp157.dtsi"
#include "stm32mp15xf.dtsi"
```

```
#include "stm32mp15-pinctrl.dtsi"
#include "stm32mp15xxac-pinctrl.dtsi"
#include "stm32mp15xx-dkx.dtsi"
#include <dt-bindings/rtc/rtc-stm32.h>

/ {
	model = "STMicroelectronics STM32MP157F-DK2 Discovery Board";
	compatible = "st,stm32mp157f-dk2", "st,stm32mp157";

	aliases {
		serial3 = &usart2;
	};

	chosen {
		stdout-path = "serial0:115200n8";
	};

	wifi_pwrseq: wifi-pwrseq {
		compatible = "mmc-pwrseq-simple";
		reset-gpios = <&gpioh 4 GPIO_ACTIVE_LOW>;
	};
};
```

6.2.1 Ficheros de cabecera

En el árbol de dispositivos se permite incluir los ficheros de cabecera con extensión *.h* con *#include* como en el lenguaje C, o bien los ficheros de árbol de dispositivo con extensión *.dtsi.*

A diferencia de los archivos *.dts* que describe la información del dispositivo a nivel de placa, *.dtsi* describe información a nivel de SoC (es decir, cuántas CPU tiene el SoC, cuál es la frecuencia principal, etc.).

Con un SoC se pueden diseñar y fabricar muchas placas de desarrollo diferentes, y la información común de estas placas se extrae como un archivo *.dtsi*, lo que permite que otros archivos hagan referencia. Por ejemplo, los ficheros como *stm32mp15-pinctrl.dtsi, stm32mp15xf.dtsi* en el código anterior son ficheros comunes de la placa stm32mp15 proporcionados por STMicroelectronics oficialmente.

6.2.2 Nodo de dispositivo

Cada dispositivo se representa por un nodo en el árbol, y el nodo se define con llaves *"{}"*. El árbol de dispositivo empieza por un raíz del árbol "*/ { ... }*" y cada árbol solo contiene un raíz. Los *"aliases { ... }"*, *"chosen { ... }"* y *"wifi_pwrseq: wifi-pwrseq { ... }"* son nodos hijos, o nodos secuendarios del raíz.

El formato de denominación de los nodos es el siguiente:

```
label: node-name@unit-address
```

Donde la parte delante del símbolo *":"* es la etiqueta del nodo, y el nombre del nodo está detrás. El objetivo de introducir la etiqueta es facilitar el acceso al nodo. Se permite acceder directamente al nodo a través de *&etiqueta*. Por ejemplo, a través de *&cpu0*, podemos acceder al nodo *"cpu@0"* sin introducir el nombre completo del nodo.

"@unit-address" se utiliza para especificar la dirección del dispositivo o la primera dirección de su registro generalmente, Y se puede omitir si el nodo no tiene una propiedad *"reg"*. Sin embargo, debe tenerse en cuenta que los nombres de los nodos del mismo nivel pueden ser los mismos, pero se requiere que la dirección de la unidad sea diferente.

6.2.3 Propiedades de nodo

El contenido entre llaves *"{}"* es la propiedad del nodo, se representan por pares clave-valor, y es la información de descripción de hardware que se pasará al kernel. Por lo general, un nodo contiene información de propiedades múltiples. Cómo escribir estas propiedades y cómo hacer referencia a estas propiedades en el driver es el punto clave en el árbol de dispositivos. En esta sección, vamos a ver algunas propiedades estándar comúnmente utilizados por muchos drivers periféricos en Linux, y que también utilizaremos en el programa.

compatible

El uso de la propiedad compatible difiere entre el nodo raíz y los nodos de dispositivo.

La propiedad compatible de dispositivos es una propiedad muy importante. El valor de la propiedad compatible es una lista de cadenas y se utiliza para vincular el dispositivo al driver en el kernel de Linux.

El formato de valor de la propiedad compatible es el siguiente:

```
"manufacturer,model"
```

Entre ellos, *manufacturer* indica el fabricante y *model* es generalmente el nombre del driver correspondiente al módulo.

Generalmente, el primer valor de la propiedad compatible del nodo raíz describe el nombre del dispositivo de hardware, y el segundo valor describe el SoC utilizado por el dispositivo. El kernel de Linux verificará si el dispositivo es compatible a través del nodo de raíz y, de ser así, el dispositivo iniciará el kernel de Linux. En los códigos de ejemplo, el nodo raíz "/" tiene la propiedad *compatible = "st, stm32mp157a-dk1", "st,*

stm32mp157", podemos saber que el nombre del dispositivo de hardware es "stm32mp157a-dk1", y el dispositivo utiliza el SoC "stm32mp157".

status

Es la información de estado del dispositivo, a través de esta propiedad, podemos deshabilitar o habilitar el dispositivo. Los estados opcionales se muestran en la siguiente tabla:

Valor	Descripción
"okay"	Indica que el dispositivo está operativo.
"disabled"	Indica que el dispositivo no está operativo actualmente, pero podría volver a funcionar en el futuro (por ejemplo, algo no está enchufado o apagado). Hay que consulte el enlace del dispositivo para obtener detalles sobre lo que significa deshabilitado para un dispositivo determinado.
"reserved"	Indica que el dispositivo está operativo, pero no debe utilizarse. Por lo general, esto se usa para dispositivos controlados por otro componente de software, como el firmware de la plataforma.
"fail"	Indica que el dispositivo no está operativo. Se detectó un error grave en el dispositivo y es poco probable que vuelva a funcionar sin reparación.
"fail-sss"	Indica que el dispositivo no está operativo. Se detectó un error grave en el dispositivo y es poco probable que vuelva a funcionar sin reparación. La parte sss del valor es específica del dispositivo e indica la condición de error detectada.

Tabla 2. Descripción de los valores opcionales de la propiedad *status* [15]

#address-cells y #size-cells

Los valores de estos dos atributos son enteros de 32 bits sin signo, se utilizan en cualquier nodo de dispositivo que tenga nodos secundarios para establecer el formato de la propiedad *reg* de su nodo secundario.

reg

Generalmente, la propiedad reg se utiliza para describir la información de la dirección del dispositivo, su formato es:

```
reg = <address1 length1 address2 length2 address3 length3……>
```

Cada combinación de *address* y *length* representa un rango de direcciones, donde *address* es la dirección inicial y *length* es la longitud de la dirección. El valor de #address-cells indica la longitud de palabra (32bits) ocupada por los datos de *address*, #size-cells indica la longitud de palabra ocupada por los datos de *length*.

6.3 Crear un nodo en el árbol de dispositivos

Para empezar, agregaremos un nodo *my_led* dentro del nodo raíz del árbol de dispositivos en el fichero *arch/arm/boot/dts/stm32mp15xx-dkx.dtsi* que soporta las placas stm32mp15. Este nodo contiene la información sobre los registros que utilizamos para encender el led LD5, que se define en su propiedad *reg*, y es similar a la definición de recursos en la sección anterior.

```
my_led {
        compatible = "my_led";
        status = "okay";
        reg = <0x50000894 0x04
                0x50000A28 0x04
                0x50002000 0x04
                0x50002018 0x04>;
};
```

Después de modificar el árbol de dispositivos, necesitamos utilizar el comando make dtbs para compilar de nuevo el árbol de dispositivos en el código fuente del kernel de Linux.

```
ine:~/STM32MPU_workspace/STM32MP1-Ecosystem-v4.0.0/files_cai/kernel/stm32mp1-ope
nstlinux-5.10-dunfell-mp1-21-11-17/sources/arm-ostl-linux-gnueabi/linux-stm32mp-
5.10.61-stm32mp-r2-r0/linux-5.10.61$ make dtbs
  DTC     arch/arm/boot/dts/stm32mp157a-dk1.dtb
  DTC     arch/arm/boot/dts/stm32mp157d-dk1.dtb
  DTC     arch/arm/boot/dts/stm32mp157c-dk2.dtb
  DTC     arch/arm/boot/dts/stm32mp157c-dk2-a7-examples.dtb
  DTC     arch/arm/boot/dts/stm32mp157c-dk2-m4-examples.dtb
  DTC     arch/arm/boot/dts/stm32mp157f-dk2.dtb
  DTC     arch/arm/boot/dts/stm32mp157f-dk2-a7-examples.dtb
  DTC     arch/arm/boot/dts/stm32mp157f-dk2-m4-examples.dtb
```

Figura 28. Resultado de compilación

Después de compilar, obtendremos los archivos *.dtb* que se permite analizar el kernel. Necesitamos copiarlos en el directorio */boot* de la placa con el comando:

scp arch/arm/boot/dts/stm32mp157*.dtb root@192.168.18.191:./boot

Y reiniciar el kernel ejecutando el comando reboot con los nuevos árboles de dispositivo empleados. Después de que Linux se inicia correctamente, podemos entrar al directorio */proc/device-tree/* para confirmar que el nuevo nodo *my_led* existe.

```
root@stm32mp1:~# ls /proc/device-tree
#address-cells  aliases     cpu0-opp-table                led                name              reserved-memory  sram@2ffff000   vin
#size-cells     arm-pmu     cpus                          memory@c0000000    pm_domain         serial-number    thermal-zones   wifi-pwrseq
__symbols__     chosen      firmware                      model              psci              soc              timer
ahb             compatible  interrupt-controller@a0021000 my_led             regulator-booster sound            usb-phy-tuning
```

Figura 29. Árbol de dispositivos en la placa

Además, las cuatro propiedades de este nodo también existen en forma de archivos, y podemos utilizar el comando cat para verificar sus valores.

```
root@stm32mp1:/sys/firmware/devicetree/base/my_led# ls
compatible  name  reg  status
root@stm32mp1:/sys/firmware/devicetree/base/my_led# cat compatible
my_ledroot@stm32mp1:/sys/firmware/devicetree/base/my_led#
```

Figura 30. Comprobación del valor de la propiedad *compatible*

Hasta ahora, hemos agregado con éxito un nodo denominado *my_led* al árbol de dispositivos de nuestro stm32mp157f-dk2.

6.4 Realización del driver empleando Árbol de dispositivos

El método de desarrollo del driver sigue siendo muy similar al anterior, porque la función principal del driver es la misma, que es registrar dispositivos, crear nodos de dispositivos, etc. Los únicos cambios son la forma en que se obtienen los recursos de registros a operar y la forma de emparejamiento con los controladores correspondientes.

6.4.1 Funciones _of

Dado que los recursos se definen en el árbol de dispositivos, en primer lugar, debemos obtener el nodo desde el árbol. Se proporciona un conjunto de funciones en kernel de Linux para obtener un nodo del árbol de dispositivos en el driver. Estas funciones están definidas en el fichero */include/Linux/of.h* del código fuente de kernel y comienzan con *of_*. Nos permiten obtener los nodos de varias formas, como buscando el nombre del nodo, buscando su nodo padre y buscando el punto del tipo de nodo etc. [17] Vamos a buscar por la ruta del nodo con la función:

```
static inline struct device_node *of_find_node_by_path(const char *path)
```

Donde el argumento *path* es la ruta completa al nodo, y devuelve un puntero a una estructura de tipo *device_node*, que representa el nodo. Vamos a añadir un nuevo miembro *device_node* para almacenar el puntero del nodo *my_led* a nuestra estructura *my_led_dev* que definimos en 4.1.

```
struct my_led_dev{
        ……
        struct device_node      *nd;    // device node
};
struct my_led_dev my_led;

        my_led.nd = of_find_node_by_path("/my_led");
        if(my_led.nd == NULL) {
                printk("my_led node not found.\r\n");
                return -EINVAL;
        } else {
                printk("my_led node found.\r\n");
        }
```

Adicionalmente, el kernel también define algunas funciones que comienzan con *_of* para obtener recursos de los nodos del dispositivo, es decir, las propiedades de los nodos. Vamos a utilizar las siguientes funciones para obtener las propiedades *compatible*, *status* y *reg* respectivamente:

of_find_property para encontrar una propiedad especificada y devuelve una estructura *property* que representa esta propiedad. De los miembros de esta estructura, podemos obtener el valor de propiedad deseado. *of_property_read_string* que lee un dato string de una propiedad, y *of_property_read_u32_array* que lee un array de tipo u32.

```
// Get value of the property compatible
prop = of_find_property(my_led.nd, "compatible", NULL);
if(prop == NULL) {
        printk("Failed to read compatible.\r\n");
} else {
        printk("compatible = %s\r\n", (char*)prop->value);
}

// Get value of the property status
err = of_property_read_string(my_led.nd, "status", &str);
if(err < 0){
        printk("Failed to read status.\r\n");
} else {
        printk("status = %s\r\n", str);
}

// Get value of the property reg
err = of_property_read_u32_array(my_led.nd, "reg", regdata, 8);
if(err < 0) {
        printk("failed to read reg.\r\n");
} else {
        for(i = 0; i < 8; i++)
                printk("reg: %#X ", regdata[i]);
        printk("\r\n");
}
```

Una vez tenemos las direcciones almacenadas en la propiedad *reg*, podemos realizar conversión entre la dirección física y la dirección virtual utilizando la función *of_iomap* en vez de *ioremap*, que es más recomendable en un sistema con árbol de dispositivos. A continuación, podemos hacer operaciones a estos registros de la misma manera definida en 4.2.1.

```
RCC_PLL4CR = of_iomap(my_led.nd, 0);
RCC_MP_AHB4ENSETR = of_iomap(my_led.nd, 1);
GPIOA_MODER = of_iomap(my_led.nd, 2);
```

```
GPIOA_BSRR = of_iomap(my_led.nd, 3);
```

6.4.2 *Emparejamiento*

Como hemos introducido anteriormente, los drivers se definen mediante la estructura *platform_driver*, que tiene miembro: driver, que es una abstracción de la estructura *device_driver*. La descripción de la estructura *device_driver* se encuentra en */include/Linux/device/driver.h*, podemos observar que tiene un miembro *of_match_table*, que es donde se almacena la tabla de coincidencias *compatible* del driver. Al registrar un dispositivo o un driver, la propiedad *compatible* de cada nodo de dispositivo en el árbol de dispositivos se comparará con todos los miembros de la tabla *of_match_table* del driver para comprobar si hay elementos idénticos. En caso de sí, significa que el dispositivo coincide con este driver.

```
struct device_driver {
        const char                *name;

        ……
        const struct of_device_id        *of_match_table;

};
```

Por lo tanto, se exige añadir el mismo valor que la propiedad *compatible* en el nodo del dispositivo que definimos, es decir, añadir *"my_led"* al *of_match_table* de nuestro driver.

```
static const struct of_device_id led_match_table[] = {
        { .compatible = "my_led" },
        { },
};

static struct platform_driver led_driver = {
        .driver         = {
                .name   = "my_led",
                .of_match_table = led_match_table,
        },
        .probe          = led_probe,
        .remove                 = led_remove,
};
```

6.5 Resultados

En este apartado, la forma de verificación es muy similar a la de la Sección 4.9 porque el dispositivo ya está añadido al árbol de dispositivos y verificado en la sección 6.3. El proceso será compilar el driver, transmitir los ficheros obtenido a la placa, cargar el driver

y utilizar los comandos definidas en *ledtest.c* para hacer operaciones, al final, descargar el driver con *rmmod* una vez finalizamos la validación.

Lo único a tener en cuenta es que cuando registramos el driver, según el log del kernel mostrado en la figura 31, el dispositivo del árbol y el driver coinciden. Además, se consigue acceder a las informaciones de registros.

```
root@stm32mp1:~/led_device_tree# insmod led_driver.ko
[ 6534.652869] Match successfully.
[ 6534.654777] my_led node found.
[ 6534.658058] compatible = my_led
[ 6534.661756] status = okay
[ 6534.663833] reg: 0X50000894
[ 6534.663843] reg: 0X4
[ 6534.666671] reg: 0X50000A28
[ 6534.668908] reg: 0X4
[ 6534.671786] reg: 0X50002000
[ 6534.674017] reg: 0X4
[ 6534.676940] reg: 0X50002018
[ 6534.679176] reg: 0X4
[ 6534.682011]
[ 6534.685935] my_led: major=242, minor=0
```

Figura 31

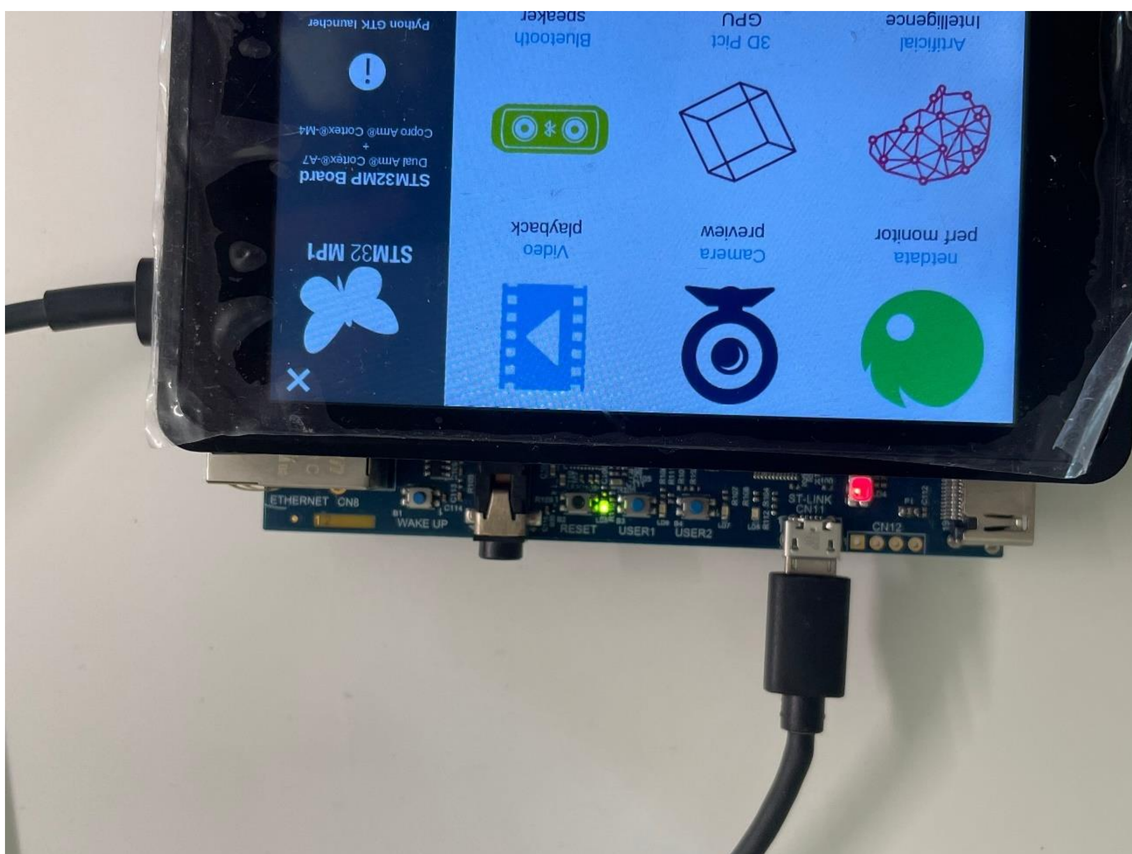

Figura 32

Capítulo 7. Conclusiones y propuesta de trabajo futuro

Comenzando con los puntos positivos, hemos alcanzado el objetivo de investigar las diferentes opciones de acceder y controlar a bajo nivel a los periféricos en un sistema Linux embebido.

Como se muestran en los resultados de los capítulos 4, 5 y 6, hemos conseguido operar los registros de la forma más simple, implementar los dispositivos y drivers de plataforma para mejorar la mantenibilidad y reutilización del código, y agregar dispositivos al árbol de dispositivos, obtener las informaciones del nodo y hacer operaciones.

En conclusión, este trabajo describe los procesos del diseño, desarrollo y verificación de drivers de dispositivo de caracteres de diferentes maneras, y contiene los conceptos básicos sobre el driver de plataforma y el árbol de dispositivo.

Desde mi punto de vista, este trabajo es como un punto de partida: nos puede servir con el fin de completar desarrollos más avanzados. Por ejemplo, las siguientes propuestas:

- Utilizar el subsistema PINCTL (PIN CONTROL) y GPIO control del sistema Linux embebido para definir GPIO en los nodos en el árbol de dispositivos, en lugar de utilizar las direcciones de los registros.
- Implementar el desarrollo de drivers para dispositivos de bloque, dispositivos de red e incluso periféricos externos conectados a la placa.
- No solo se limita a la operación del hardware, sino también realizar desarrollo de drivers que involucran conceptos como los principios del sistema operativo, el manejo de interrupciones y el control de concurrencia.

Capítulo 8. Bibliografía

[1] STMicroelectronics, «STM32MPU Embedded Software distribution,» [En línea]. Available: https://wiki.st.com/stm32mpu/wiki/STM32MPU_Embedded_Software_distribution.

[2] STMicroelectronics, «Which STM32MPU Embedded Software Package better suits your needs,» [En línea]. Available: https://wiki.st.com/stm32mpu/wiki/Which_STM32MPU_Embedded_Software_Package_better_suits_your_needs.

[3] STMicroelectronics, «STM32MP1 Developer Package,» [En línea]. Available: https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package#Developer_Package_content.

[4] A. T.-H. D.-E. J. G. M.-C. D. J. G.-A. AdrianIglesias-Benitez1*, «Desarrollo de un driver GNU/Linux para sistemas de adquisición de datos embebidos,» *Revista Cubana de Ciencias Informáticas,* vol. 8, nº 2, pp. 35-51, 2014.

[5] S. L. Gabriel, «Desarrollo de un Device Driver de Linux y su Librería de espacio usuario para gestionar Redes de Petri Generalizadas en el ke,» 2019.

[6] J. C. Alessandro Rubini, Linux device drivers: Second Edition, 2001.

[7] STMicroelectronics, «User manual, Discovery kits with STM32MP157 MPUs,» [En línea]. Available: https://www.st.com/resource/en/user_manual/dm00591354-discovery-kits-with-stm32mp157-mpus-stmicroelectronics.pdf.

[8] STMicroelectronics, «LEDs and buttons on STM32 MPU boards,» [En línea]. Available: https://wiki.st.com/stm32mpu/wiki/LEDs_and_buttons_on_STM32_MPU_boards.

[9] STMicroelectronics, «STM32MP157x-DKx motherboard schematics,» [En línea]. Available: https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group0/36/8e/ea/7a/ca/ca/4b/e4/mb1272-dk2-c01_schematic/files/MB1272-DK2-C01_Schematic.pdf/jcr:content/translations/en.MB1272-DK2-C01_Schematic.pdf.

[10] STMicroelectronics, «STM32MP157 reference manual,» [En línea]. Available: https://www.st.com/resource/en/reference_manual/DM00327659-.pdf.

[11] oracle, «Device Driver Tutorial,» [En línea]. Available: https://docs.oracle.com/cd/E19253-01/817-5789/index.html.

[12] A. L. d. l. Ríos, Linux Driver Development for Embedded Processors - Second Edition, 2018.

[13] «The Linux Kernel documentation, Linux Device Model,» [En línea]. Available: https://linux-kernel-labs.github.io/refs/heads/master/labs/device_model.html.

[14] «The Linux Kernel documentation, Platform Devices and Drivers,» [En línea]. Available: https://www.kernel.org/doc/html/latest/driver-api/driver-model/platform.html#platform-devices-and-drivers.

[15] devicetree.org, «Devicetree Specification, Release v0.3-40-g7e1cc17,» 2021.

[16] Octavo Systems, «OSD335x Lesson 2: Linux Device Tree,» [En línea]. Available: https://octavosystems.com/app_notes/osd335x-design-tutorial/osd335x-lesson-2-minimal-linux-boot/linux-device-tree/.

[17] «The Linux Kernel, DeviceTree Kernel API,» [En línea]. Available: https://docs.kernel.org/devicetree/kernel-api.html.

Anexo I. Código fuente del proyecto realizado en el capítulo 4

led_drv.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/mutex.h>
#include <linux/wait.h>
#include <linux/uaccess.h>
#include <asm/io.h>
#include <linux/device.h>
#include <asm/uaccess.h>
#include <linux/types.h>
#include <linux/ide.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <asm/mach/map.h>

#define LEDON                   1
#define LEDOFF                  0
#define LED_COUNT  1
#define LED_NAME    "myled"
#define LED_CLASS_NAME   "myled_class"

#define RCC_BASE_ADDR                   (0x50000000)
#define GPIOA_BASE_ADDR                 (0x50002000)
#define RCC_PLL4CR_ADDR                 (RCC_BASE_ADDR + 0x894)
#define RCC_MP_AHB4ENSETR_ADDR          (RCC_BASE_ADDR + 0xA28)
#define GPIOA_MODER_ADDR                (GPIOA_BASE_ADDR + 0x00)
#define GPIOA_BSRR_ADDR                 (GPIOA_BASE_ADDR + 0x18)

static volatile unsigned int *RCC_PLL4CR;
```

```
static volatile unsigned int *RCC_MP_AHB4ENSETR;
static volatile unsigned int *GPIOA_MODER;
static volatile unsigned int *GPIOA_BSRR;

// led information structure
struct my_led_dev {
	dev_t dev_nb;			// device number
	int major;			// major device number
	int minor;			// minor device number
	struct cdev cdev;		// cdev
	struct class *class;		// class
	struct device *device;		// device
};
struct my_led_dev my_led;

void led_unmap(void) {
	iounmap(RCC_PLL4CR);
	iounmap(RCC_MP_AHB4ENSETR);
	iounmap(GPIOA_MODER);
	iounmap(GPIOA_BSRR);
}

static int led_open(struct inode *inode, struct file *filp) {
	// Enalbe PLL4
	*RCC_PLL4CR |= (1<<0);
	while ((*RCC_PLL4CR & (1<<1)) == 0);

	// Enable gpioA for A7 core
	*RCC_MP_AHB4ENSETR |= (1<<0);

	// Set PA14 to output mode
	*GPIOA_MODER &= ~(3<<28);	// PA14, set bit[29:28] to 0x00
	*GPIOA_MODER |= (1<<28);	// Set bit[29:28] to 01
	return 0;
}
```

```
static ssize_t led_write(struct file *filp, const char __user *buf,
                                size_t count, loff_t *ppos) {
        char val;   // led status
        int ret;
        ret = copy_from_user(&val, buf, 1);
        if (ret < 0) {
                printk(KERN_ERR "led_write: failed to copy %d\n", ret);
                return ret;
        }

        // led control
        if (val == LEDON) {
                *GPIOA_BSRR = (1<<30);
        }
        else if (val == LEDOFF) {
                *GPIOA_BSRR = (1<<14);
        }
        return 0;
}

static int led_close(struct inode *inode, struct file *filp) {
        return 0;
}

static struct file_operations led_simple_ops = {
        .owner          = THIS_MODULE,
        .write          = led_write,
        .open           = led_open,
        .release   = led_close,
};

static int __init led_init(void)
{
        int ret;
        printk("%s %s %d\n", __FILE__, __FUNCTION__, __LINE__);
```

```
	// 0x50000000 + 0x894
	RCC_PLL4CR = ioremap(RCC_PLL4CR_ADDR, 4); // 4 bytes
	// 0x50000000 + 0xA28
	RCC_MP_AHB4ENSETR = ioremap(RCC_MP_AHB4ENSETR_ADDR, 4);
	// 0x50002000 + 0x00
	GPIOA_MODER = ioremap(GPIOA_MODER_ADDR, 4);
	// 0x50002000 + 0x18
	GPIOA_BSRR = ioremap(GPIOA_BSRR_ADDR, 4);

	// device number
	if (my_led.major) {	// if major is defined
		my_led.dev_nb = MKDEV(my_led.major, 0);
		ret = register_chrdev_region(my_led.dev_nb, LED_COUNT, LED_NAME);
		if(ret < 0) {
			pr_err("register_chrdev_region failed on %s, ret=%d\r\n",
LED_NAME, ret);
			led_unmap();
		}
	} else {
		ret = alloc_chrdev_region(&my_led.dev_nb, 0, LED_COUNT,
LED_NAME);
		if(ret < 0) {
			pr_err("alloc_chrdev_region failed on %s, ret=%d\r\n", LED_NAME,
ret);
			led_unmap();
		}
		my_led.major = MAJOR(my_led.dev_nb);
		my_led.minor = MINOR(my_led.dev_nb);
	}
	printk("my_led: major=%d, minor=%d\r\n", my_led.major, my_led.minor);
	// cdev
	my_led.cdev.owner = THIS_MODULE;
	cdev_init(&my_led.cdev, &led_simple_ops);
	ret = cdev_add(&my_led.cdev, my_led.dev_nb, LED_COUNT);
	if(ret < 0){
		unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
	}
```

```
	// create class
	my_led.class = class_create(THIS_MODULE, LED_CLASS_NAME);
	if (IS_ERR(my_led.class)) {
		cdev_del(&my_led.cdev);
		return -1;
	}
	// create device
	my_led.device = device_create(my_led.class, NULL, my_led.dev_nb, NULL,
LED_NAME);	//dev/myled
	if (IS_ERR(my_led.device)) {
		class_destroy(my_led.class);
		return -1;
	}
	return 0;
}

static void __exit led_exit(void) {
	// led unmap
	led_unmap();
	device_destroy(my_led.class, my_led.dev_nb);
	class_destroy(my_led.class);
	cdev_del(&my_led.cdev);
	unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
}
module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
```

Makefile

```
KERN_DIR = <KERNEL PATH>

obj-m	+= led_drv.o

all:
	make -C $(KERN_DIR) M=`pwd` modules
	$(CROSS_COMPILE)gcc -o ledtest ledtest.c
```

```
clean:
	make -C $(KERN_DIR) M=`pwd` modules clean
	rm -rf modules.order
	rm -f ledtest

deploy:
	scp *.ko root@<board ip address>:./led_simple
	scp ledtest root@<board ip address>:./led_simple
```

Anexo II. Código fuente del proyecto realizado en el capítulo 5

led_device.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <linux/of_gpio.h>
#include <linux/semaphore.h>
#include <linux/timer.h>
#include <linux/irq.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <linux/fcntl.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define RCC_BASE_ADDR                     (0x50000000)
#define GPIOA_BASE_ADDR                   (0x50002000)
```

```
#define RCC_PLL4CR_ADDR                 (RCC_BASE_ADDR + 0x894)
#define RCC_MP_AHB4ENSETR_ADDR          (RCC_BASE_ADDR + 0xA28)
#define GPIOA_MODER_ADDR                (GPIOA_BASE_ADDR + 0x00)
#define GPIOA_BSRR_ADDR                 (GPIOA_BASE_ADDR + 0x18)
#define REGISTER_LENGTH                 4

static void led_dev_release(struct device *dev) {
	printk("Led device released.\r\n");
}

static struct resource led_resources[] = {
	[0] = {
			.start	= RCC_PLL4CR_ADDR,
			.end	= (RCC_PLL4CR_ADDR + REGISTER_LENGTH - 1),
			.flags	= IORESOURCE_MEM,
		},
		[1] = {
			.start	= RCC_MP_AHB4ENSETR_ADDR,
			.end	= (RCC_MP_AHB4ENSETR_ADDR + REGISTER_LENGTH - 1),
			.flags	= IORESOURCE_MEM,
		},
		[2] = {
			.start	= GPIOA_MODER_ADDR,
			.end	= (GPIOA_MODER_ADDR + REGISTER_LENGTH - 1),
			.flags	= IORESOURCE_MEM,
		},
		[3] = {
			.start	= GPIOA_BSRR_ADDR,
			.end	= (GPIOA_BSRR_ADDR + REGISTER_LENGTH - 1),
			.flags	= IORESOURCE_MEM,
		},
};

static struct platform_device led_device = {
	.name = "my_led",
	.dev = {
```

```
            .release = led_dev_release,
          },
        .num_resources = ARRAY_SIZE(led_resources),     // number of resources
        .resource = led_resources,                      // resource
};

static int __init led_dev_init(void) {
    return platform_device_register(&led_device);
}

static void __exit led_dev_exit(void) {
    platform_device_unregister(&led_device);
}

module_init(led_dev_init);
module_exit(led_dev_exit);
MODULE_LICENSE("GPL");
```

led_driver.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>
#include <linux/platform_device.h>
#include <linux/types.h>
```

```
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <linux/of_gpio.h>
#include <linux/semaphore.h>
#include <linux/timer.h>
#include <linux/irq.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <linux/fcntl.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define LEDON              1
#define LEDOFF             0
#define LED_COUNT          1
#define LED_NAME    "myled"
#define LED_CLASS_NAME   "myled_class"

static int led_pins[4];
static volatile unsigned int *RCC_PLL4CR;
static volatile unsigned int *RCC_MP_AHB4ENSETR;
static volatile unsigned int *GPIOA_MODER;
static volatile unsigned int *GPIOA_BSRR;

// led information structure
struct my_led_dev {
        dev_t dev_nb;               // device number
        int major;                  // major device number
        int minor;                  // minor device number
        struct cdev cdev;           // cdev
        struct class *class;        // class
        struct device *device;      // device
};
```

```
struct my_led_dev my_led;

void led_unmap(void) {
	iounmap(RCC_PLL4CR);
	iounmap(RCC_MP_AHB4ENSETR);
	iounmap(GPIOA_MODER);
	iounmap(GPIOA_BSRR);
}

static int led_open(struct inode *inode, struct file *filp) {
	// Enalbe PLL4
	*RCC_PLL4CR |= (1<<0);
	while ((*RCC_PLL4CR & (1<<1)) == 0);
	// Enable gpioA for A7 core
	*RCC_MP_AHB4ENSETR |= (1<<0);
	// Set PA14 to output mode
	*GPIOA_MODER &= ~(3<<28);	// PA14, set bit[29:28] to 0x00
	*GPIOA_MODER |= (1<<28);	// Set bit[29:28] to 01

	return 0;
}

static ssize_t led_write(struct file *filp, const char __user *buf,
				size_t count, loff_t *ppos) {
	char val;	// led status
	int ret;
	ret = copy_from_user(&val, buf, 1);
	if (ret < 0) {
		printk(KERN_ERR "led_write: failed to copy %d\n", ret);
		return ret;
	}
	// led control
	if (val == LEDON) {
		*GPIOA_BSRR = (1<<30);
	}
	else if (val == LEDOFF) {
```

```
                *GPIOA_BSRR = (1<<14);
        }
        return 0;
}

static struct file_operations led_platform_ops = {
        .owner          = THIS_MODULE,
        .write          = led_write,
        .open           = led_open,
};

static int led_probe(struct platform_device *dev)          // init
{
        int i = 0, ret;
        struct resource *led_resource;

        printk("Match successfully.\r\n");

        // Get the resource
        for (i = 0; i < 4; i++) {
     led_resource = platform_get_resource(dev, IORESOURCE_MEM, i);

                if (!led_resource) {
                        printk("Failed to get resource.\r\n");
                        return -1;
                }
     led_pins[i] = led_resource->start;
        }

        RCC_PLL4CR = ioremap(led_pins[0], 4);
        RCC_MP_AHB4ENSETR = ioremap(led_pins[1], 4);
        GPIOA_MODER = ioremap(led_pins[2], 4);
        GPIOA_BSRR = ioremap(led_pins[3], 4);

        // device number
        if (my_led.major) {   // if major is defined
```

```
		my_led.dev_nb = MKDEV(my_led.major, 0);
		ret = register_chrdev_region(my_led.dev_nb, LED_COUNT, LED_NAME);
		if(ret < 0) {
			pr_err("register_chrdev_region failed on %s, ret=%d\r\n", LED_NAME, ret);
			led_unmap();
		}
	} else {
		ret = alloc_chrdev_region(&my_led.dev_nb, 0, LED_COUNT, LED_NAME);
		if(ret < 0) {
			pr_err("alloc_chrdev_region failed on %s, ret=%d\r\n", LED_NAME, ret);
			led_unmap();
		}
		my_led.major = MAJOR(my_led.dev_nb);
		my_led.minor = MINOR(my_led.dev_nb);
	}
	printk("my_led: major=%d, minor=%d\r\n", my_led.major, my_led.minor);
	// cdev
	my_led.cdev.owner = THIS_MODULE;
	cdev_init(&my_led.cdev, &led_platform_ops);
	ret = cdev_add(&my_led.cdev, my_led.dev_nb, LED_COUNT);
	if(ret < 0){
		unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
	}
	// create class
	my_led.class = class_create(THIS_MODULE, LED_CLASS_NAME);
	if (IS_ERR(my_led.class)) {
		cdev_del(&my_led.cdev);
		return -1;
	}
	// create device
	my_led.device = device_create(my_led.class, NULL, my_led.dev_nb, NULL, LED_NAME);
	if (IS_ERR(my_led.device)) {
		class_destroy(my_led.class);
		return -1;
```

```
        }
        return 0;
}

// remove platform_device
static int led_remove(struct platform_device *dev) {
        led_unmap();
        device_destroy(my_led.class, my_led.dev_nb);
        class_destroy(my_led.class);
        cdev_del(&my_led.cdev);
        unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
        return 0;
}

static struct platform_driver led_driver = {
        .driver         = {
                .name   = "my_led",
        },
        .probe          = led_probe,
        .remove         = led_remove,
};

static int __init led_init(void) {
    return platform_driver_register(&led_driver);
}

static void __exit led_exit(void) {
    platform_driver_unregister(&led_driver);
}

module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
```

Makefile

```
KERN_DIR =  <KERNEL PATH>
```

```
obj-m += led_device.o led_driver.o

all:
	make -C $(KERN_DIR) M=`pwd` modules
	$(CROSS_COMPILE)gcc -o ledtest ledtest.c

clean:
	make -C $(KERN_DIR) M=`pwd` modules clean
	rm -rf modules.order
	rm -f ledtest

deploy:
	scp *.ko root@<board ip address>:./led_platform
	scp ledtest root@<board ip address>:./led_platform
```

Anexo III. Código fuente del proyecto realizado en el capítulo 6

<kernel path>\linux\arch\arm\boot\dts\stm32mp15xx-dkx.dtsi

```
/ {
        my_led {
                compatible = "my_led";
                status = "okay";
                reg = <0x50000894 0x04
                        0x50000A28 0x04
                        0x50002000 0x04
                        0x50002018 0x04>;
         };
        ……
}
```

led_driver.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/gpio.h>
```

```
#include <linux/cdev.h>
#include <linux/of_gpio.h>
#include <linux/semaphore.h>
#include <linux/timer.h>
#include <linux/irq.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <linux/fcntl.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/of.h>
#include <linux/of_address.h>

#define LEDON                   1
#define LEDOFF                  0
#define LED_COUNT  1
#define LED_NAME    "myled"
#define LED_CLASS_NAME  "myled_class"

static volatile unsigned int *RCC_PLL4CR;
static volatile unsigned int *RCC_MP_AHB4ENSETR;
static volatile unsigned int *GPIOA_MODER;
static volatile unsigned int *GPIOA_BSRR;

// led information structure
struct my_led_dev {
        dev_t dev_nb;                   // device number
        int major;                      // major device number
        int minor;                      // minor device number
        struct cdev cdev;               // cdev
        struct class *class;            // class
        struct device *device;          // device
        struct device_node      *nd;    // device node
};
struct my_led_dev my_led;
```

```
void led_unmap(void) {
        iounmap(RCC_PLL4CR);
        iounmap(RCC_MP_AHB4ENSETR);
        iounmap(GPIOA_MODER);
        iounmap(GPIOA_BSRR);
}

static int led_open(struct inode *inode, struct file *filp) {
        // Enalbe PLL4
        *RCC_PLL4CR |= (1<<0);
        while ((*RCC_PLL4CR & (1<<1)) == 0);
        // Enable gpioA for A7 core
        *RCC_MP_AHB4ENSETR |= (1<<0);
        // Set PA14 to output mode
        *GPIOA_MODER &= ~(3<<28);      // PA14, set bit[29:28] to 0x00
        *GPIOA_MODER |= (1<<28);       // Set bit[29:28] to 01
        return 0;
}

static ssize_t led_write(struct file *filp, const char __user *buf,
                        size_t count, loff_t *ppos) {
        char val;  // led status
        int ret;
        ret = copy_from_user(&val, buf, 1);
        if (ret < 0){
                printk(KERN_ERR "led_write: failed to copy %d\n", ret);
                return ret;
        }
        // led control
        if (val == LEDON) {
                *GPIOA_BSRR = (1<<30);
        }
        else if (val == LEDOFF) {
                *GPIOA_BSRR = (1<<14);
        }
```

```
        return 0;
}

static struct file_operations led_platform_ops = {
        .owner          = THIS_MODULE,
        .write          = led_write,
        .open           = led_open,
};

static int led_probe(struct platform_device *dev)         // init
{
        int i = 0, err;
        struct property *prop;
        const char *str;
        u32 regdata[12];
        printk("Match successfully.\r\n");
        // Get the device node
        my_led.nd = of_find_node_by_path("/my_led");
        if(my_led.nd == NULL) {
                printk("my_led node not found.\r\n");
                return -EINVAL;
        } else {
                printk("my_led node found.\r\n");
        }
        // Get value of the property compatible
        prop = of_find_property(my_led.nd, "compatible", NULL);
        if(prop == NULL) {
                printk("Failed to read compatible.\r\n");
        } else {
                printk("compatible = %s\r\n", (char*)prop->value);
        }
        // Get value of the property status
        err = of_property_read_string(my_led.nd, "status", &str);
        if(err < 0) {
                printk("Failed to read status.\r\n");
        } else {
```

```
		printk("status = %s\r\n", str);
	}
	// Get value of the property reg
	err = of_property_read_u32_array(my_led.nd, "reg", regdata, 8);
	if(err < 0) {
		printk("failed to read reg.\r\n");
	} else {
		for(i = 0; i < 8; i++)
			printk("reg: %#X ", regdata[i]);
		printk("\r\n");
	}

	RCC_PLL4CR = of_iomap(my_led.nd, 0);
	RCC_MP_AHB4ENSETR = of_iomap(my_led.nd, 1);
	GPIOA_MODER = of_iomap(my_led.nd, 2);
	GPIOA_BSRR = of_iomap(my_led.nd, 3);
	// device number
	if (my_led.major) {   // if major is defined
		my_led.dev_nb = MKDEV(my_led.major, 0);
		err = register_chrdev_region(my_led.dev_nb, LED_COUNT, LED_NAME);
		if(err < 0) {
			pr_err("register_chrdev_region failed on %s, ret=%d\r\n", 
LED_NAME, err);
			led_unmap();
		}
	} else {
		err = alloc_chrdev_region(&my_led.dev_nb, 0, LED_COUNT, 
LED_NAME);
		if(err < 0) {
			pr_err("alloc_chrdev_region failed on %s, ret=%d\r\n", LED_NAME, 
err);
			led_unmap();
		}
		my_led.major = MAJOR(my_led.dev_nb);
		my_led.minor = MINOR(my_led.dev_nb);
	}
	printk("my_led: major=%d, minor=%d\r\n", my_led.major, my_led.minor);
```

```
	// cdev
	my_led.cdev.owner = THIS_MODULE;
	cdev_init(&my_led.cdev, &led_platform_ops);
	err = cdev_add(&my_led.cdev, my_led.dev_nb, LED_COUNT);
	if(err < 0) {
		unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
	}
	// create class
	my_led.class = class_create(THIS_MODULE, LED_CLASS_NAME);
	if (IS_ERR(my_led.class)) {
		cdev_del(&my_led.cdev);
		return -1;
	}
	// create device
	my_led.device = device_create(my_led.class, NULL, my_led.dev_nb, NULL,
LED_NAME);
	if (IS_ERR(my_led.device)) {
		class_destroy(my_led.class);
		return -1;
	}
	return 0;
}

// remove platform_device
static int led_remove(struct platform_device *dev) {
	led_unmap();
	device_destroy(my_led.class, my_led.dev_nb);
	class_destroy(my_led.class);
	cdev_del(&my_led.cdev);
	unregister_chrdev_region(my_led.dev_nb, LED_COUNT);
	return 0;
}

static const struct of_device_id led_match_table[] = {
	{ .compatible = "my_led" },
	{ },
};
```

```
static struct platform_driver led_driver = {
        .driver         = {
                .name   = "my_led",
                .of_match_table = led_match_table,
        },
        .probe          = led_probe,
        .remove         = led_remove,
};

static int __init led_init(void) {
    return platform_driver_register(&led_driver);
}

static void __exit led_exit(void) {
    platform_driver_unregister(&led_driver);
}

module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
```

Makefile

```
KERN_DIR =  <KERNEL PATH>

obj-m += led_driver.o

all:
        make -C $(KERN_DIR) M=`pwd` modules
        $(CROSS_COMPILE)gcc -o ledtest ledtest.c

clean:
        make -C $(KERN_DIR) M=`pwd` modules clean
        rm -rf modules.order
        rm -f ledtest
```

```
deploy:
	scp *.ko root@<board ip address>:./led_device_tree
	scp ledtest root@<board ip address>:./led_device_tree
```

Anexo IV. Código fuente de la aplicación de verificación

ledtest.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
/* Usage:
ledtest /dev/myled on
ledtest /dev/myled off
*/
int main(int argc, char **argv) {
	int fd, ret;
	char status = 0;
	if (argc != 3) {
		printf("Usage: %s <dev> <on|off>\n", argv[0]);
		printf("  eg: %s /dev/myled on\n", argv[0]);
		printf("  eg: %s /dev/myled off\n", argv[0]);
		return -1;
	}
	// open
	fd = open(argv[1], O_RDWR);
	if (fd < 0) {
		printf("File not found: %s\n", argv[1]);
		return -1;
	}
	// write
	if (strcmp(argv[2], "on") == 0) {		// led on
		status  = 1;
	}
	ret = write(fd, &status, 1);
	if(ret < 0) {
		printf("Failed to control LED.\r\n");
		close(fd);
		return -1;
```

```
        }
        ret = close(fd);
        if(ret < 0) {
                printf("Failed to close file.\r\n");
                return -1;
        }
        return 0;
}
```