UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería Electrónica

DISEÑO DE DRIVERS PARA CONTROL DEL HARDWARE DE UN SISTEMA EMBEBIDO BASADO EN LINUX

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Sistemas Electrónicos

AUTOR/A: Soriano Viguer, Esther

Tutor/a: Torres Carot, Vicente

CURSO ACADÉMICO: 2021/2022

Resumen

Este trabajo tiene como finalidad el diseño y desarrollo de drivers que permitan controlar distintos elementos hardware conectados a un sistema embebido basado en el sistema operativo Linux. Para alcanzar los objetivos es necesario conocer cómo el sistema operativo gestiona el acceso al hardware, además de los distintos tipos de drivers que existen en Linux.

La solución desarrollada se basa en diferentes drivers que se ejecutan en el espacio del sistema operativo, y permiten su fácil uso desde la capa de usuario.

Resum

Aquest treball té com a finalitat el disseny i desenvolupament de 'drivers' que permeten controlar diferents elements hardware connectats a un sistema embegut basat en el sistema operatiu Linux. Per a aconseguir els objectius és necessari conéixer com el sistema operatiu gestiona l'accés a l'hardware, a més dels diferents tipus de 'drivers' que existeixen en Linux.

La solució desenvolupada es basa en diferents 'drivers' que s'executen en l'espai del sistema operatiu, i permeten el seu fàcil ús des de la capa d'usuari.

Abstract

The aim of this project is the Linux drivers' design and development to be able to control some hardware elements connected to an embedded system working with Linux. In order to achieve the project's goals it is needed to know how the operating system handles hardware access, as well as learning about the different kinds of drivers that exists into Linux world.

Implemented solution is based on different types of drivers which work in kernel space and which allow their easy usage through user space.

Índice general

Índice de figuras

Índice de tablas

Listado de siglas empleadas

API Application Programming Interfaces.

DT Device tree.

GPIO General Purpose Input Output.

LCD Lyquid Crystal Display.

LED Light Emitting Diode.

MMU Memory Management Unit.

SDK Software Development Kit.

SSH Secure Shell.

Parte I

Memoria

Capítulo 1

Introducción

1.1. Motivación

El uso de los sistemas embebidos se encuentra en constante crecimiento y es que estos han pasado a formar parte de nuestro día a día. Es posible encontrarlos en una gran infinidad de dispositivos, como son teléfonos móviles, infraestructuras de redes, automóviles, juguetes, señalización digital, escaneos médicos y muchísimas otras aplicaciones industriales y de consumo. El sistema operativo por excelencia para estos sistemas embebidos es Linux, debido a la gran flexibilidad que ofrece, así como por su estabilidad y su licencia de código abierto.

Es por todo lo anterior por lo que es interesante comprender cómo el sistema operativo interactúa con el hardware, lo que a bajo nivel se simplificaría en cómo se puede realizar una lectura y una escritura sobre un registro. Sin embargo, también es interesante conocer las distintas capas de abstracción de las que dispone Linux para unificar y simplificar el acceso a todos los tipos de periféricos que soporta.

1.2. Objetivos

Como objetivo global a perseguir durante el proyecto se tiene el de aprender a desarrollar controladores (drivers) que permitan interactuar con elementos hardware (LEDs, LCDs, puertos serie, etc) conectados a un sistema embebido basado en Linux.

Como objetivos específicos se tienen:

- Estudio del sistema operativo Linux y cómo este interactúa con el hardware.
- Elección de los tipos de drivers a desarrollar basándose en los tipos que existen y su clasificación.
- Diseño y desarrollo de cada uno de los drivers para el control de los elementos hardware que se deseen.

- Verificación de los drivers con el hardware.

1.3. Estructura del documento

El desarrollo del trabajo sigue la siguiente disposición. En el capítulo 2 se realiza un análisis de los posibles tipos de drivers que existen dentro del entorno de Linux, así como se exponen conocimientos teóricos sobre este sistema operativo para poder comprender el posterior desarrollo. A continuación, en el capítulo 3, se explica el diseño y el desarrollo de los drivers realizados, entrando en detalle en cada uno de ellos. En el capítulo 4 se muestran los resultados obtenidos de los drivers desarrollados, analizando en profundidad que cada uno de ellos cumple las funcionalidades esperadas.

Para terminar, en el capítulo 5 se exponen las conclusiones del trabajo, analizando el cumplimiento de los objetivos propuestos y en el capítulo 6 se proponen algunas actividades que continuarían con la línea de trabajo del proyecto. Cabe destacar que en los anexos se incluye el código desarrollado de los drivers y algunas hojas de datos usadas.

Capítulo 2

Descripción de las soluciones estudiadas

En este capítulo se expondrán los diferentes recursos existentes para la realización del proyecto, así como se explicarán detalles más teóricos relacionados con el entorno de Linux, con la finalidad de conocer y seleccionar la manera más adecuada de desarrollar drivers de periféricos.

2.1. Introducción a Linux

En el siguiente apartado se dará una breve introducción al sistema operativo Linux, tanto a nivel de historia como a nivel técnico. Todo esto se realiza con el objetivo de poder comprender todo el desarrollo de los drivers en esta plataforma, ya que implica un conocimiento medio en relación a este sistema operativo.

2.1.1. Un poco de historia

Linux es un sistema operativo de código abierto desarrollado inicialmente para ordenadores con arquitectura Intel-x86. Sin embargo, ha ido evolucionado y migrándose a una gran cantidad de plataformas hardware diferentes, desde sistemas embebidos a superordenadores.

Linus Torvalds empezó en 1991 a desarrollar su propio kernel de sistema operativo, así como otros elementos necesarios para construir un sistema operativo completo con su kernel en el centro. Más tarde esto se conocería como el kernel de Linux. En 1992, Linux obtuvo la *Licencia Pública General de GNU*[1], conocida como *GPL*, que declara que el software es libre, por lo que permite su uso, distribución y modificación, así como lo protege de intentos de apropiación. Tras la obtención de esta licencia se hizo posible construir una comunidad de desarrolladores de todas partes del mundo. Esto hizo que a mediados de los años 90, tras combinar el kernel de Linux con otros componentes del proyecto GNU, apareciesen sistemas completos conocidos como *distribuciones Linux*.

Estas distribuciones tuvieron una gran repercusión en el movimiento del software de código abierto, y tras esto, grandes empresas como IBM y Oracle anunciaron su apoyo a la plataforma Linux, así como destinaron grandes esfuerzos de desarrollo.

Hoy en día, Linux se ejecuta en más de la mitad de servidores de Internet, en la gran mayoría de *smartphones* y otros muchos sistemas embebidos, así como en los superordenadores más potentes del mundo [2].

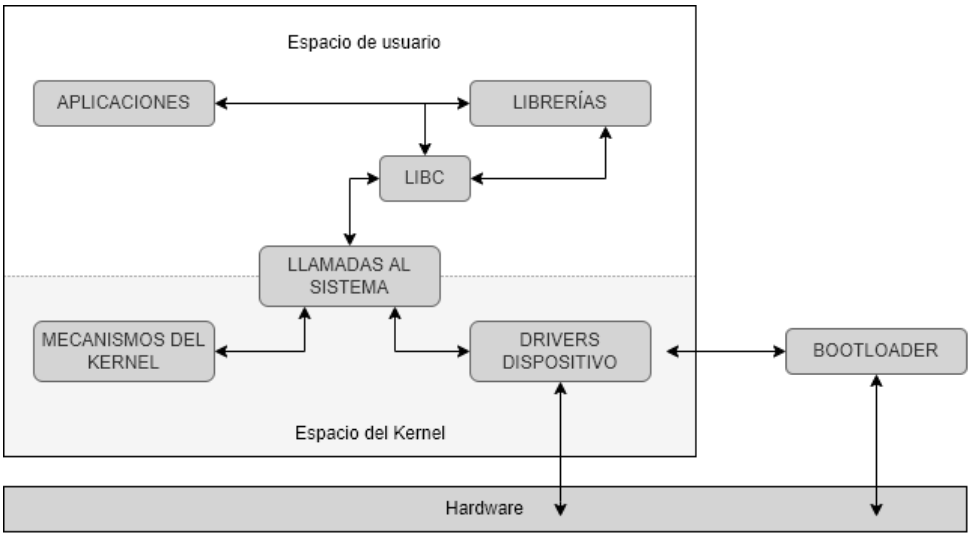


Figura 2.1: Arquitectura a alto nivel de un sistema embebido Linux

2.1.2. Información técnica

Los componentes principales de un sistema embebido Linux son los siguientes:

- **Bootloader** (o "gestor de arranque")

  Se trata de un pequeño programa escrito en código máquina específico que sirve para inicializar el sistema operativo. Se encarga de configurar el sistema de memoria, cargar la imagen del kernel y del *device tree* (árbol de dispositivos), iniciar la línea de comandos y otros elementos necesarios al inicio. Existen diferentes opciones, aunque el *bootloader* estándar para ARM Linux es *U-boot*.

- **Kernel** (o "núcleo")

  El Kernel de Linux es el nivel más bajo de software corriendo en un sistema Linux. Entre sus funciones se encuentran el manejo del hardware, la ejecución de aplicaciones de usuario y el mantenimiento de la seguridad e integridad de todo el sistema. Constituye el núcleo del sistema Linux, por lo que su funcionamiento resulta determinante. Este kernel es el resultado de uno de los mayores proyectos de software cooperativos, en el que cada tres meses aproximadamente se libera una nueva versión estable.

- **Interfaz de llamadas al sistema** (o *System call interface*)

  Las llamadas al sistema constituyen la interfaz fundamental entre una aplicación y el kernel de Linux, y son el único modo en el que una aplicación de usuario (corriendo en el espacio de usuario) puede interactuar con el kernel (espacio del kernel), tal y como se observa en la Figura 2.1. Esta separación tan estricta entre espacio de usuario y espacio de kernel provoca que las aplicaciones de usuario no puedan acceder libremente a los recursos internos del kernel, lo que garantiza la seguridad y estabilidad del sistema.

- **C-Runtime library** (o "librería C en tiempo de ejecución")

  La librería C en tiempo de ejecución define macros, definiciones de tipos, funciones para el manejo de funciones matemáticas, caracteres y texto, procesado de ficheros, asignación de memoria y muchas otras funciones que dependen del sistema operativo. Como se observa en la figura 2.1, cuando se ejecuta una llamada al sistema se hace mediante funciones de más alto nivel encapsuladas precisamente en esta librería C. También destacar que existen numerosas librerías C en tiempo de ejecución, pero la librería C por excelencia es *glibc*, perteneciente al GNU.

- **Librerías compartidas del sistema**

  Las librerías compartidas del sistema son librerías que son cargadas por los programas cuando estos se inician. Normalmente estas librerías se enlazan con las aplicaciones de usuario para proveer acceso a funcionalidades específicas del sistema, entre las que se encuentran el acceso al hardware. En otras palabras, estas librerías se encargan de encapsular la funcionalidad del sistema y son una parte fundamental a la hora de construir aplicaciones que interactúan con el sistema.

- **Root filesystem** (o "sistema de archivos raíz")

  El sistema de archivos raíz es donde se almacenan todos los ficheros de la jerarquía del sistema de archivos (esto es librerías, aplicaciones, datos del sistema, etc.). Linux organiza la estructura de carpetas del *rootfs* de acuerdo al estándar FHS (*Filesystem-Hierarchy- Standard*)[3]. Este estándar define los nombres, permisos y ubicaciones para muchos tipos de ficheros y directorios. Además asegura la compatibilidad entre diferentes distribuciones de Linux. En la figura 2.2 se puede observar un ejemplo del sistema de archivos raíz de un sistema Linux.

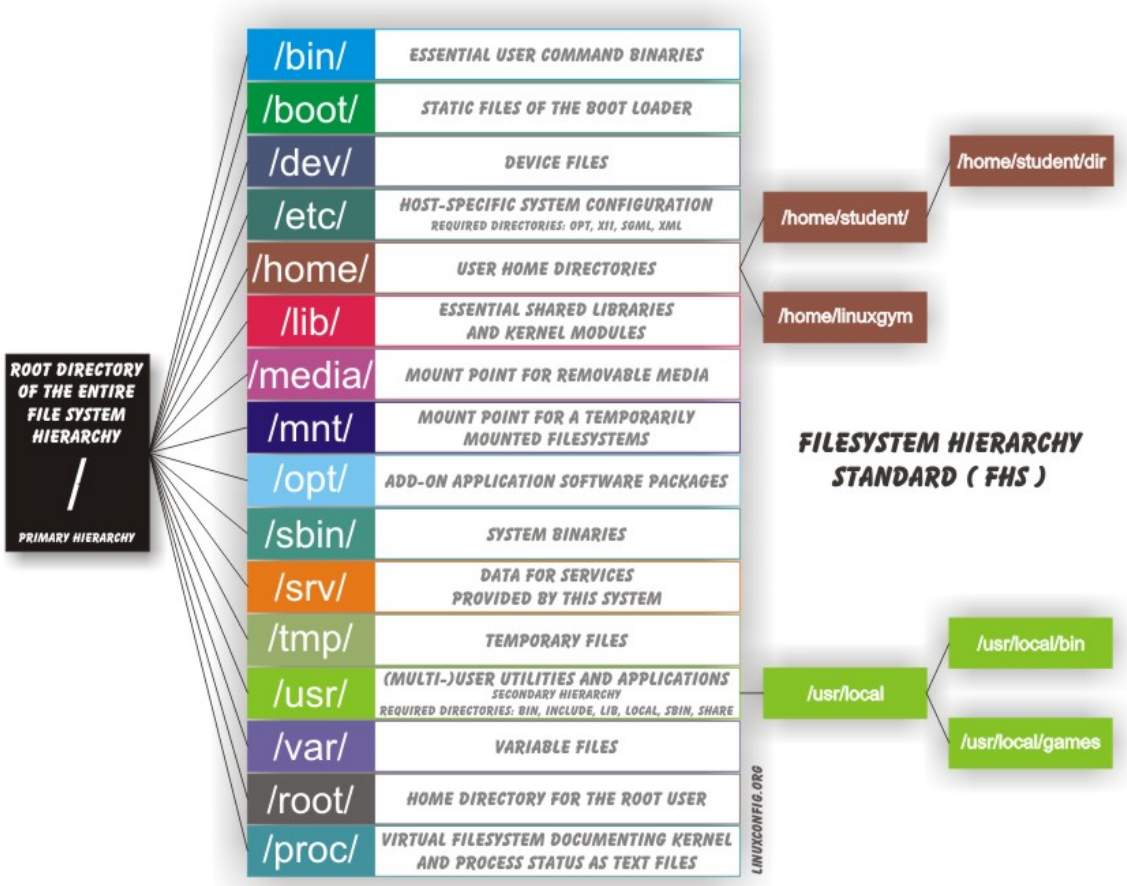


Figura 2.2: Esquema del sistema de archivos raíz de un sistema Linux

2.2. Drivers de dispositivo en Linux

Un *driver* (o driver de dispositivo) es una pieza de software cuyo propósito es el de controlar y gestionar un dispositivo hardware concreto. Desde el punto de vista de un sistema operativo, un driver puede estar en el espacio del kernel (funcionando en modo privilegiado) o en el espacio de usuario (con privilegios menores).

A continuación se van a enumerar las ventajas y desventajas de cada tipo de driver.

Drivers de kernel	
Ventajas	**Inconvenientes**
Se ejecutan en el espacio del kernel con el modo privilegiado más alto, por lo que tienen acceso a todos los recursos del hardware y manejo de interrupciones	El desarrollo de estos drivers resulta más complicado para los desarrolladores, debido a que la API del kernel es diferente y más extensa que la API del nivel de aplicación
La API que ofrece el kernel al espacio de usuario permite a múltiples aplicaciones acceder al driver del kernel simultáneamente	Los errores de código o bugs pueden terminar causando que el kernel termine repentinamente su ejecución (crash)
La mayoría de drivers suelen están en el espacio del kernel, por lo que existe más soporte al respecto	La API del kernel está expuesta a constantes cambios, por lo que los drivers hechos para una versión de kernel pueden no funcionar para otra
	Son más complicados de depurar ya que el código del kernel está muy optimizado

Tabla 2.1: Ventajas y desventajas de los drivers de kernel.

Drivers de usuario	
Ventajas	**Inconvenientes**
Más fáciles de depurar y más herramientas de depuración disponibles	No tienen acceso a las infraestructuras del kernel, ni a sus servicios
El acceso a los dispositivos es muy eficiente ya que no hay llamadas al sistema	No es posible manejar interrupciones (sin usar un driver de kernel que notifique al espacio de usuario, lo que causa retardos)
La API de Linux es muy estable	No existe una API predefinida para permitir a las aplicaciones acceder al driver
Se puede escribir el driver en cualquier lenguaje, no solo en C	

Tabla 2.2: Ventajas y desventajas de los drivers de usuario.

Tras analizar todas las ventajas y desventajas de los drivers de kernel y de los drivers de usuario y teniendo en cuenta el objetivo principal del proyecto, se van a desarrollar principalmente drivers de dispositivo a nivel de kernel. Esta elección se basa en el propósito de conocer cómo el sistema operativo interactúa con el hardware y gestiona sus recursos. Por todo lo anterior, resulta también interesante aprender a gestionar interrupciones, y de otro modo no sería posible explorar esta funcionalidad.

2.2.1. Clasificación de dispositivos y drivers

Para transferir datos desde el espacio del kernel al espacio de usuario, Linux usa los *device nodes* o nodos de dispositivo, que también son conocidos como ficheros virtuales. Estos ficheros existen en el sistema de archivos raíz de Linux, aunque no son ficheros como tal. Cuando un usuario lee de un nodo de dispositivo, el sistema operativo copia esos datos capturados por el driver del dispositivo correspondiente al espacio de memoria de la aplicación. Del mismo modo, cuando se escribe en un nodo de dispositivo, el sistema operativo copia esos datos proporcionados por la aplicación, al driver del dispositivo. Estos ficheros virtuales se pueden 'abrir' y 'cerrar', así como ser 'escritos' y 'leídos', todo esto haciendo uso de las llamadas estándar del sistema.

La forma de ver de Linux a los dispositivos, distingue entre tres tipos de dispositivos fundamentales: dispositivo de carácter (o *char device*), dispositivo de bloque (o *block device*) y dispositivo de red (o *network device*). Cada driver generalmente implementa uno de estos tipos, y por lo tanto es clasificable en estos mismos grupos. El concepto de todos los tipos de dispositivo es el mismo, aunque se distinguen principalmente en la forma en el que se accede a ellos. A continuación se detallan cada uno de estos dispositivos.

- **Dispositivos de carácter**

  Son los dispositivos más comunes, y son aquellos a los que se puede acceder como un flujo de bytes o un archivo. Un driver de char se encarga de implementar este

comportamiento, implementando normalmente las llamadas de sistema de apertura, cierre, lectura y escritura. La consola de texto y los puertos serie son ejemplos de este tipo de dispositivos. Se accede a los mismos mediante nodos de dispositivo del sistema de archivos, como `/dev/tty0`.

- **Dispositivos de bloque**

  Como los dispositivos char, los nodos del sistema de archivos acceden a los dispositivos de bloques sobre el directorio `/dev`. Un dispositivo de bloque es un dispositivo (por ejemplo, un disco) que puede alojar un sistema de archivos. Los dispositivos de bloque y char difieren solo en la forma en que los datos son administrados internamente por el kernel. Al igual que un dispositivo char, a cada dispositivo de bloque se accede a través de un nodo del sistema de archivos, y la diferencia entre ellos es transparente para el usuario.

- **Dispositivos de red**

  Estos dispositivos son accedidos a través de la interfaz de sockets BSD y los subsistemas de red. Resultan más complejos y no van a ser tratados durante el desarrollo del proyecto.

Cabe destacar que hay otras formas de clasificar a los drivers compatibles con los tipos de dispositivos anteriores. En general, algunos tipos de controladores funcionan con capas adicionales de funciones de soporte del kernel para un tipo determinado de dispositivo. Por ejemplo, se puede hablar de módulos de bus serie universal (USB), módulos serie, módulos SCSI, etc. Todos los dispositivos USB se controlan mediante un módulo USB que funciona con el subsistema USB, pero el dispositivo aparece en el sistema como un dispositivo char (un puerto serie USB, por ejemplo), un dispositivo de bloque (un lector de tarjetas de memoria USB), o un dispositivo de red (una interfaz Ethernet USB).

2.3. Gestión de los dispositivos en Linux

Linux usa la tecnología 'Plug and Play' que ofrece soporte para agregar y eliminar automáticamente dispositivos del sistema. Esta tecnología reduce los conflictos con los recursos usados, al configurarse automáticamente al inicio del sistema. Para poder soportar esta tecnología, es necesario poder detectar automáticamente la adición y eliminación de dispositivos del sistema, así como poder gestionar los recursos de los dispositivos. También resulta necesario que los dispositivos permitan la configuración del software, además de que el sistema operativo pueda automáticamente cargar los drivers que correspondan cuando sea necesario. Finalmente, se requiere poder añadir y eliminar dispositivos del sistema mientras este está en ejecución, sin tener que reiniciarlo.

Para poder cumplir los requisitos anteriormente mencionados, y poder hacer uso de esta tecnología 'Plug and Play', Linux desarrolló un modelo para dispositivos Linux : *Linux Device Model*, o modelo de dispositivos Linux.

La comprensión de este modelo es crucial para el desarrollo de drivers en este sistema operativo. Los modelos de dispositivo y de driver son maneras universales de organizar los dispositivos y los drivers en buses.

Este sistema tiene una gran cantidad de ventajas, como son:

- La reducción de código duplicado.
- La organización del código y la separación de los drivers de dispositivo de los drivers de controlador, la separación de la descripción del hardware de los drivers en sí mismos, etc.
- Permite la capacidad de determinar todos los dispositivos del sistema, consultar sus estados, ver a qué bus pertenecen y cual es su driver correspondiente.
- Permite la construcción de un *device tree* ('árbol de dispositivos') que describe la estructura completa de todos los dispositivos del sistema, incluyendo todas las interconexiones y buses.
- Permite el emparejamiento de dispositivos con sus drivers, y viceversa.
- Permite la clasificación de dispositivos según su tipo, como dispositivos de entrada, sin la necesidad de conocer la tipología del dispositivo físico.

Este modelo contiene los términos siguientes:

- **Dispositivo**: un objeto físico o virtual que está enlazado a un bus.
- **Driver**: una entidad software asociada con un dispositivo y que realiza operaciones con este, así como realiza ciertas funciones de gestión.
- **Bus**: un dispositivo que sirve como punto de conexión para otros dispositivos.
- **Clase**: un tipo de dispositivo que comparte un funcionamiento similar. Existen clases para particiones, discos, puertos series, etc.
- **Subsistema**: una vista a la estructura del sistema. Los subsistemas del kernel incluyen dispositivos, buses, clases, etc.

El modelo de dispositivos se organiza en torno a las siguientes estructuras de datos, que garantizan la interacción entre un dispositivo de hardware y un controlador de dispositivo (o driver).

1. La estructura *bus_type* que representa un tipo de bus (por ejemplo, USB, I2C, PCI).
2. La estructura *device_driver* que representa a un driver capaz de gestionar dispositivos en un bus.
3. La estructura *device* que representa a un dispositivo conectado a un bus.

2.3.1. Dispositivos plataforma

Dentro del modelo de dispositivos de Linux, existe un conjunto de dispositivos llamados *dispositivos plataforma*, que son aquellos que normalmente aparecen como entidades autónomas del sistema. La mayoría de controladores integrados en un SoC (controladores de UART, controladores de Ethernet, controladores de SPI, etc) forman parte de este grupo, que usa un bus especial, denominado *bus de plataforma* que accede directamente a la CPU.

Estos dispositivos no pueden ser detectados dinámicamente, sino que se han de describir estáticamente. Esto se hace a través de un fichero de descripción del hardware, denominado *device tree* o **árbol de dispositivos**, que se lee en el arranque del sistema operativo. El uso de este fichero permite además la separación entre los drivers y el hardware, permitiendo realizar drivers genéricos sin vincularlos a pines concretos y asociarlos directamente a los pines correspondientes mediante parámetros en el *device tree*. Los controladores que se realizan para este tipo de dispositivos se denominan *platform drivers*.

2.3.1.1. Asociación driver - dispositivo físico

Este proceso, también conocido como *binding* permite asociar un dispositivo con su correspondiente driver (del tipo *platform driver*) mediante el identificador del dispositivo (que corresponde con una cadena de caracteres).

Cuando se carga el módulo del driver en el kernel, el driver del dispositivo de plataforma se registra a sí mismo en el driver del bus de plataforma, haciendo uso de la función `platform_device_register()`. Es entonces cuando se evalúa el *device tree* y se comprueba si alguno de los dispositivos de ese fichero corresponde con alguno de los nombres de los dispositivos compatibles indicados en el driver. Esto se ampliará en secciones posteriores inlcuyendo ejemplos.

Si se encuentra en el árbol de dispositivos un dispositivo compatible con el driver, el driver de bus empareja el dispositivo con su driver y se inicializa el dispositivo.

2.4. Gestión de la memoria en Linux

2.4.1. Memoria física y memoria virtual

Linux usa un sistema de memoria virtual, lo que significa que las direcciones usadas por los programas no corresponden directamente con direcciones físicas usadas por el hardware. Esto tiene numerosas ventajas, entre las que destacan el hecho de que con el uso de la memoria virtual, los programas pueden alojar más memoria de que existe físicamente; así como simplifica el uso de la memoria, ya que la memoria física no está necesariamente contigua y sus rangos de acceso cambian mucho entre arquitecturas.

Por tanto, el kernel y los procesos de usuario usan direcciones virtuales y existe un elemento hardware denominado **MMU** (unidad de manejo de memoria) que se encarga de traducir las direcciones virtuales a las físicas, proteger el acceso a memoria, controlar la caché, etc. En la arquitectura ARM, esta traducción se define a través de unas tablas de traducción almacenadas en memoria que la MMU lee automáticamente cuando lo necesita.

La memoria física se divide en páginas, el tamaño de las cuales depende de la arquitectura. Además cada una de estas páginas puede ser mapeada como una o más páginas virtuales.

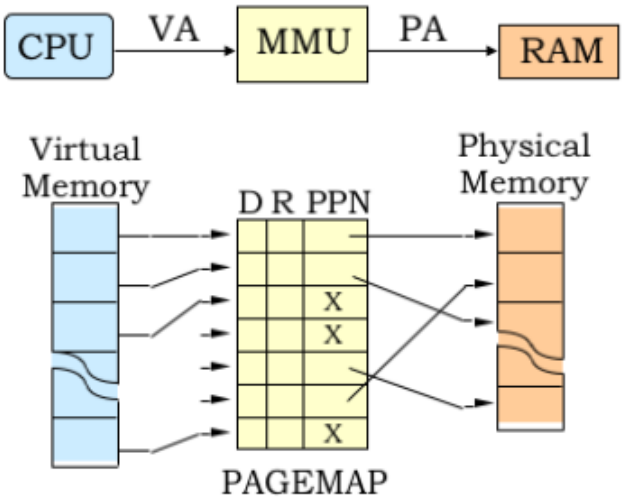


Figura 2.3: Esquema del acceso a memoria en Linux

2.4.2. Mapeo de memoria en drivers

El mapeo de memoria es una de las utilidades más interesantes de los sistemas Unix. Para un driver de dispositivo, la utilidad del mapeo de memoria permite acceso directo a memoria a un dispositivo en la capa de usuario. Para poder mapear memoria, un driver tiene que definir esta operación en su estructura de operaciones `file_operations()`, y de este modo, en la capa de usuario, sobre el nodo de dispositivo correspondiente se podrá hacer uso de la llamada al sistema `mmap()`.

Capítulo 3

Diseño y desarrollo de drivers

En el siguiente capítulo se explicará el desarrollo realizado de distintos tipos de drivers a nivel de kernel. En primer lugar, se expondrá el entorno de desarrollo usado así como sus requisitos principales, para dar paso posterior a las diferentes explicaciones de las implementaciones realizadas.

3.1. Requisitos iniciales

Para poder realizar drivers a nivel de kernel en Linux son necesarios los siguientes elementos:

- **Ficheros fuente del kernel** de Linux de la versión para la que se desee desarrollar los drivers. Es importante destacar que dado que los drivers hacen uso de la API del kernel, cambios en el kernel pueden hacer que los drivers no resulten compatibles.
- **Herramientas necesarias de compilación o compilación cruzada**. Una vez el código del driver se encuentre terminado, se habrá de compilar para obtener el binario que se cargará en el sistema deseado. Si la compilación se realiza en una arquitectura diferente de la arquitectura destino se denomina compilación cruzada.
- Algún método para **transferir el binario** del driver generado al sistema donde se cargará.

En este proyecto se va a hacer uso del kit de desarrollo STM32MP157C-DK2 del fabricante *STMicroelectronics*. Esta placa contiene, entre otros, un procesador ARM Cortex-A7 y usa Linux como sistema operativo. El fabricante de la misma ofrece una distribución de su software con soporte para tres tipos distintos de usuario, en función de la configurabilidad que se desee obtener. Es por todo lo anterior, por lo que se ha considerado una gran opción para el proyecto y se ha decidido hacer uso de la misma.

De entre los paquetes distribuidos por el fabricante, tras informarse sobre los perfiles objetivo de cada uno de ellos, se ha escogido el *Developer Package* ya que ofrece la versatilidad para cambiar el kernel de Linux y es el recomendado para el desarrollo de drivers.

3.2. Entorno de desarrollo

La distribución escogida del software proporcionado por el fabricante contiene una herramienta SDK para realizar la compilación cruzada, además de todos los ficheros fuente del kernel de Linux, U-boot, etc. Siguiendo todos los pasos descritos [4] por el fabricante es posible configurar el kernel de Linux, realizar su compilación cruzada e incluirlo en la placa de desarrollo. Es importante destacar que el fabricante recomienda inicialmente compilar el kernel y distribuirlo en la placa antes de desarrollar drivers por primera vez. Esto se debe al hecho de que la versión de kernel compilada y distribuida en la placa ha de ser la misma que se use para compilar los drivers.

La versión del kernel de Linux que se va a emplear en los desarrollos corresponde con la versión 5.10.61. Con respecto a la configuración del kernel, la que viene por defecto del fabricante sirve como base para los drivers a realizar, a excepción de que es necesario habilitar la funcionalidad de *Userspace IO driver* para poder realizar el driver UIO.

Continuando con el entorno de desarrollo, para realizar los drivers de kernel se va a hacer uso del lenguaje de programación C. Para la compilación, además del SDK proporcionado que contiene el compilador cruzado y todas las configuraciones y dependencias, se hace uso de *make*, creando ficheros *Makefile* para organizar las dependencias del código y describir las órdenes a ejecutar para obtener y distribuir los binarios a partir del código fuente. Por tanto, cada driver que se implemente tendrá su correspondiente *Makefile*, o un único *Makefile* que contenga las instrucciones para todos los drivers.

También cabe destacar que para distribuir los binarios de los drivers en la placa se va hacer uso de un cliente SSH, teniendo que conectar la placa a la red mediante el puerto Ethernet.

3.3. Introducción al desarrollo de drivers

El kernel de Linux es modular, lo que significa que está formado por módulos o fragmentos de código que pueden ser insertados o eliminados con el objetivo de añadir o quitar una funcionalidad. Los drivers a nivel de kernel se implementan como módulos. En Linux, estos módulos tienen extensión '.ko' y se pueden cargar o descargar en el sistema operativo en tiempo de ejecución mediante comandos como `insmod` o `rmmod`.

Cabe destacar que para el desarrollo de los siguientes drivers ha resultado de gran ayuda el libro *Linux Driver Development for Embedded Processors*[5].

3.4. Driver dummy

El primer driver que se va a desarrollar no va a tener otro propósito que el de introducirse en los controladores de dispositivo y poder entender como interactúan con el resto de elementos del sistema.

Como se ha comentado, los drivers de kernel se implementan por medio de módulos del kernel, por lo que inicialmente se va a crear un módulo *dummy*. A partir de aquí se

irán añadiendo modificaciones hasta convertirlo en un driver de dispositivo, aunque sin ninguna actuación sobre el hardware.

3.4.1. Creación de un módulo *dummy*

Como es sabido, los módulos pueden ser cargados en el sistema operativo en tiempo de ejecución. Cuando esto ocurre, se invoca a la función descrita en `module_init`. Para el caso descrito en el siguiente Listing 3.1, al cargar el módulo se ejecutará la función `helloworld_init`, que mostrará en la consola el mensaje descrito.

Cuando se elimina el módulo cargado, el sistema operativo llama a la función descrita en `module_exit`, en este caso `helloworld_exit`, que mostrará otro mensaje.

Listing 3.1: dummy_module.c

```
#include <linux/module.h>

static int __init helloworld_init(void)
{
    pr_info("Hello world\n");
    return 0;
}

static void __exit helloworld_exit(void)
{
    pr_info("End of the world\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);

MODULE_AUTHOR("Esther Soriano");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple helloworld module");
```

3.4.2. Creación de un char driver *dummy*

A continuación se va a implementar el primer *char driver* , o driver de carácter. Como se ha comentado en el apartado 2.2.1, a este tipo de dispositivos se accede a través de sus nodos de dispositivo, usando su correspondiente driver.

3.4.2.1. Operaciones de un char driver

Desde el punto de vista de una aplicación, un dispositivo de carácter se abre haciendo la llamada al sistema `open()`, se cierra con `close()` y se puede operar con él con otras llamadas como `read()`, `write()` o `ioctl()`. Para esto, el driver correspondiente tiene que implementar estas operaciones en una estructura del tipo `struct file_operations`, declarada en el fichero cabecera `include/linux/fs.h`.

De esta manera, cuando se realiza la correspondiente llamada al sistema desde el espacio de usuario, la capa del sistema de archivos de Linux asegura que las operaciones del

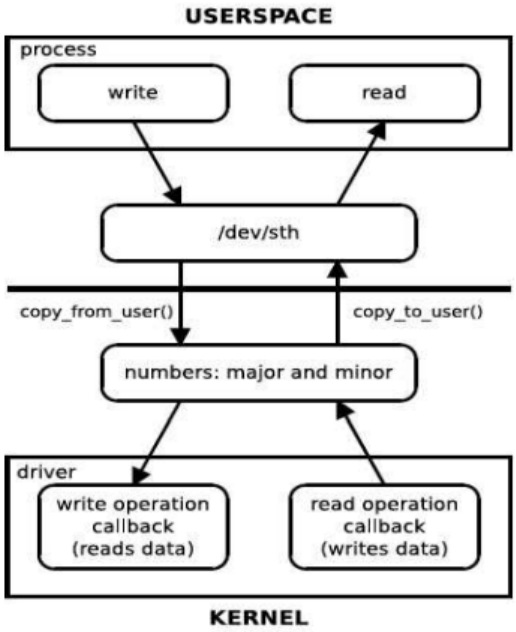


Figura 3.1: Esquema de la interacción en un driver de carácter

driver son llamadas. Desde el espacio del kernel, el driver implementa y registra todas las operaciones de vuelta. Esta interacción se observa en la figura 3.1, en la que también se puede apreciar que el kernel hace uso de su API con funciones como `copy_from_user()` o `copy_to_user()` para intercambiar información con la capa de usuario. Otras funciones similares son `put_user()` o `get_user()`.

En el siguiente Listing 3.2 se implementan las operaciones de apertura, cierre, lectura, escritura e interacción (`ioctl`) con el dispositivo. La única acción que realizan las operaciones de apertura, cierre e interacción en este ejemplo, es la de mostrar un mensaje con la acción que se está realizando sobre el dispositivo. Sin embargo, la operación de escritura permite almacenar el mensaje recibido, y la operación de lectura, devuelve este mensaje cuando es llamada. También se observa en el ejemplo la creación de la estructura `struct file_operations` y como se inicializa haciendo referencia a las funciones de nuestro driver.

Listing 3.2: char_driver.c

```
...
#include <linux/fs.h>
#define MAX_LENGTH 20U

char msg[MAX_LENGTH];

static int my_dev_open(struct inode *inode, struct file *file)
{
  pr_info("my_dev_open() is called.\n");
  return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
```

```
{
  pr_info("my_dev_close() is called.\n");
  return 0;
}

static ssize_t my_dev_read(struct file *file, char __user *buff, size_t count,
    loff_t *loff)
{
  ssize_t len = min(MAX_LENGTH - *loff, count);

  if(len <= 0)
    return 0;

  if(copy_to_user(buff, msg + *loff, len))
  {
    pr_info("Bad value copied");
    return -EFAULT;
  }

  pr_info("my_dev_read() is called with value %s \n", msg);

  *loff += len;
  return len;
}


static int my_dev_write(struct file *file, const char __user *buff, size_t count,
    loff_t *ppos)
{
    // Borra el contenido anterior de la variable msg.
  memset(msg, 0, MAX_LENGTH);

  if(count > MAX_LENGTH)
    count = MAX_LENGTH;

  if(copy_from_user(msg, buff, count))
  {
    pr_info("Bad value copied");
    return -EFAULT;
  }
  pr_info("Copied %d bytes from user\n", count);

  pr_info("my_dev_write() is called with value %s \n", msg);
  return count;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
  pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
  return 0;
}

// Declaración de la estructura file_operations del driver
static const struct file_operations my_dev_fops = {
  .owner    = THIS_MODULE,
  .open     = my_dev_open,
  .release    = my_dev_close,
  .read    = my_dev_read,
```

```
72    .write   = my_dev_write,
      .unlocked_ioctl  = my_dev_ioctl,
};
...
```

3.4.2.2. Creación, identificación y acceso a los dispositivos

En Linux, cada dispositivo se identifica con dos números: uno mayor y otro menor. Cada driver de dispositivo registra el número mayor en el kernel, y es responsable de gestionar el número menor. Cuando se accede a un dispositivo, el kernel usa el número mayor para identificar al driver que ha de ejecutar la operación, mientras que el número menor depende del dispositivo y se maneja internamente en el driver. La obtención de estos identificadores numéricos ha de hacerse a través de las funciones descritas en la API del kernel para este propósito, con el objetivo asegurar que estos no están en uso.

Como se ha mencionado, desde el espacio de usuario se usan los nodos de dispositivo para acceder a estos, creándose normalmente en el directorio */dev*. La creación de estos nodos anteriormente se realizaba de manera manual, a partir de los identificadores numéricos de los dispositivos mediante el comando `mknod`. Más tarde, surgieron otros métodos que automatizaban el proceso, como son el *devtmpfs* y el *miscellaneous framework*.

En este primer driver de carácter se van a realizar dos implementaciones, una haciendo uso del *devtmpfs*, y otra usando el *miscellaneous framework*, debido a que es interesante conocer ambos. Para el resto de drivers a realizar se usará el segundo método, con el objetivo de simplificar los drivers.

3.4.2.3. Implementación

A continuación se van a listar y explicar brevemente los pasos a realizar en la inicialización del driver, con el objetivo de poder entender y seguir el código que se muestra en el Listing 3.4.

1. **Asignar los identificadores numéricos del dispositivo a crear.**

   Como se ha mencionado anteriormente, los identificadores numéricos se han de obtener del kernel, a través de su API. Esto garantiza que los identificadores no se encuentran en uso. La función *alloc_chrdev_region* permite obtener del kernel el número de identificadores de dispositivo que se requiera.

2. **Inicializar la estructura del dispositivo a crear.**

   Se ha de crear la estructura del dispositivo (del tipo *cdev*) e inicializarla con la estructura que define las operaciones que realizará el driver, vista anteriormente.

3. **Registrar el dispositivo en el kernel.**

4. **Crear una clase para el dispositivo creado.**

   Como se desea crear el nodo de dispositivo usando *devtempfs* se ha de crear una clase para el dispositivo. Esta será visible en el directorio */sys/class/*.

5. **Crear el nodo del dispositivo**.

   Finalmente, se ha de crear el nodo de dispositivo que estará accesible en */dev*.

Del mismo modo, cuando se requiera destruir el driver, se habrá de ir eliminando todo lo que se haya creado hasta el momento. Es decir, si el driver ha terminado de inicializarse, se habrá de eliminar el nodo del dispositivo, eliminar la clase del dispositivo creado, eliminar la estructura del dispositivo y liberar los identificadores numéricos del dispositivo (como se observa, se realiza en el orden contrario al de creación). Cabe destacar que cuando se produce algún error en alguno de los pasos de la inicialización del driver, también se han de liberar todos los recursos usados hasta el momento. Esto se observará más adelante en el código 3.4.

A nivel de desarrollo, para crear el driver se van a hacer uso de diferentes funciones de la API del kernel. A continuación en el Listing 3.3 se muestra la definición de cada una de ellas.

Listing 3.3: kernel API

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
     const char *name);

void unregister_chrdev_region(dev_t from, unsigned count);

void cdev_init(struct cdev *cdev, const struct file_operations *fops);
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
void cdev_del(struct cdev *p);

struct class *class_create(struct module *owner, const char *name);
void class_destroy(struct class *cls);

struct device *device_create(struct class *class, struct device *parent,
         dev_t devt, void *drvdata, const char *fmt, ...);
void device_destroy(struct class *class, dev_t devt);
```

Cabe destacar que también se hacen uso de otras funciones del kernel como son `pr_info` para gestionar la información de los errores; `IS_ERR` para comprobar si ha habido algún error; o las macros `MAJOR` y `MKDEV`, para obtener los identificadores numéricos de los dispositivos del kernel.

Listing 3.4: dummy_char_driver.c

```
[...]
#include <linux/device.h>
#include <linux/cdev.h>

#define DEVICE_NAME "mydev"
#define CLASS_NAME "my_dummy_class"

static struct class* dummyClass;
static struct cdev my_dev;
dev_t dev;

static int __init init(void)
{
```

```
  int ret;
  dev_t dev_no;
16 int Major;
  struct device* helloDevice;

  pr_info("Hello dummy driver init\n");
20
  // 1. Asignación dinámica de los identificadores de los dispositivos.
  // En este caso solo se requiere un único dispositivo, que tendrá el minor 0.
  ret = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);
24 if (ret < 0){
    pr_info("Unable to allocate Major number \n");
    return ret;
  }
28
  // Obtener los identificadores asignados.
  Major = MAJOR(dev_no);
  dev = MKDEV(Major,0);
32
  pr_info("Allocated correctly with major number %d\n", Major);

    // 2. Inicialización de la estructura cdev con las operaciones que realizará el driver.
36  cdev_init(&my_dev, &my_dev_fops);

  // 3. Registro del dispositivo creado en el kernel
  ret = cdev_add(&my_dev, dev, 1);
40 if (ret < 0){
    unregister_chrdev_region(dev, 1);
    pr_info("Unable to add cdev\n");
    return ret;
44 }

  // 4. Crear y registrar una clase para el dispositivo creado
    // (accesible en /sys/class/CLASS_NAME).
48 dummyClass = class_create(THIS_MODULE, CLASS_NAME);
  if (IS_ERR(dummyClass)){
    unregister_chrdev_region(dev, 1);
    cdev_del(&my_dev);
52  pr_info("Failed to register device class\n");
    return PTR_ERR(dummyClass);
  }

56 pr_info("device class registered correctly\n");

  // 5. Crear un nodo de dispositivo en /dev/DEVICE_NAME
  // asociado al dispositivo creado.
60 helloDevice = device_create(dummyClass, NULL, dev, NULL, DEVICE_NAME);
  if (IS_ERR(helloDevice)){
      class_destroy(dummyClass);
      cdev_del(&my_dev);
64    unregister_chrdev_region(dev, 1);
      pr_info("Failed to create the device\n");
      return PTR_ERR(helloDevice);
  }
68 pr_info("The device is created correctly\n");
  return 0;
}
```

```
72 static void __exit exit(void)
   {
      // Eliminar el dispositivo.
    device_destroy(dummyClass, dev);
76
    // Destruir la clase del dispositivo.
    class_destroy(dummyClass);

80  // Eliminar la estructura del dispositivo.
    cdev_del(&my_dev);

    // Liberar los identificadores del dispositivo.
84  unregister_chrdev_region(dev, 1);

    pr_info("Dummy driver exit\n");
   }
88 [...]
```

Combinando los Listings 3.2 y 3.4, se obtiene un driver de dispositivo funcional que muestra en pantalla las operaciones del dispositivo que son llamadas. Al cargar el módulo se crea un dispositivo, accesible en `/dev/mydev`, del que se puede leer y escribir desde el espacio de usuario (por la terminal de Linux o mediante una aplicación). En el Listing A.2 del Anexo A se puede observar el código completo del driver.

3.4.2.4. Implementación alternativa usando *misc framework*

A continuación se va realizar un driver con la misma funcionalidad pero haciendo uso del *misc framework* que ofrece Linux. Se trata de una interfaz que provee el kernel de Linux para que los módulos registren el número menor de sus identificadores numéricos.

Al contrario que en la implementación anterior en la que se obtenían los dos identificadores numéricos del kernel, si se hace uso del *misc framework*, el número mayor será siempre el asociado a los *miscellaneous devices* (número 10). Con respecto al número menor, se usa normalmente una asignación dinámica para gestionarlo, aunque se puede escoger un número concreto. El uso de esta interfaz elimina por tanto posibles conflictos con otros drivers, así como simplifica la gestión.

Otra diferencia significativa con respecto a la implementación anterior, es el hecho de que al crear el dispositivo como un *misc* device, pertenece a esta misma clase, y la entrada creada en el *sysfs* será añadida en el directorio `/sys/class/misc`. Sin embargo, el nodo de dispositivo seguirá en el directorio`/dev/mydev`.

En el siguiente Listing 3.5 se observa parte de la API del kernel para gestionar *misc devices*. Se puede apreciar la propia estructura que define a un dispositivo de esta clase, así como dos funciones para registrar y dar de baja a estos dispositivos.

Listing 3.5: misc api

```
struct miscdevice {
    int minor;
4   const char* name;
    const struct file_operations *fops;
```

```
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};

int misc_register(struct miscdevice *misc);
void misc_deregister(struct miscdevice *misc);
```

A continuación, en el Listing 3.6 se muestran únicamente los cambios con respecto a la implementación anterior, haciendo uso de las funciones anteriormente descritas.

Como se observa, al hacer uso de la clase *misc*, se simplifica mucho el proceso de inicialización del driver, ya que al registrar un dispositivo de la clase *miscdevice*, no hay que preocuparse de la asignación de los identificadores numéricos, ni de la creación de una clase específica ni del nodo de dispositivo, ya que todo esto se encuentra garantizado. Lo mismo ocurre cuando se da de baja al dispositivo. Es por esto por lo que se usará esta interfaz en los drivers siguientes.

Listing 3.6: dummy_char_driver_misc.h

```
...
#include <linux/miscdevice.h>

#define DEVICE_NAME "mydev"
...

static struct miscdevice my_misc_device = {
  .minor = MISC_DYNAMIC_MINOR,
  .name = DEVICE_NAME,
  .fops = &my_dev_fops,
};

static int __init init(void)
{
  int ret;
  pr_info("Hello world init\n");

  // Registrar el misc device en el kernel
  ret = misc_register(&my_misc_device);
  if (ret != 0){
     pr_err("Failed to register the misc device\n");
     return ret;
  }
  pr_info("misc device registered correctly\n");
  return 0;
}

static void __exit exit(void)
{
   // Eliminar el misc device
  misc_deregister(&my_misc_device);
  pr_info("Dummy driver exit\n");
}
...
```

3.5. Driver GPIO (LED)

A continuación se va a implementar un driver de carácter para controlar los leds de usuario de los que dispone la placa STM32MP157C-DK2. Existen una gran cantidad de mecanismos para implementar este driver, ya que el propio sistema Linux dispone de varias interfaces a nivel de GPIOs y de LEDs que pueden ser usadas para abstraerse del hardware y realizar múltiples funcionalidades complejas. Sin embargo, el objetivo del desarrollo del driver es el de poder controlar el hardware al nivel más bajo posible, por lo que se va a implementar un driver que maneja el GPIO a nivel de sus registros.

En primer lugar, antes de entrar en la implementación específica del driver, es necesario conocer la información física del hardware que se desea controlar. En concreto, se van a usar los LEDs de usuario de los que dispone la placa STM32MP157C-DK2. Como se observa en la Figura 3.2 los LEDs verde y rojo se encuentran conectados a los pines 14 y 13 del GPIOA, mientras que el azul se encuentra en el pin 11 del GPIOD. El LED naranja no se usará en las implementaciones siguientes.

LED interface

Table 13 describes the I/O configuration of the LED interface.

Table 13. I/O configuration of the LED interfaces

I/O	Configuration
PA14	PA14 is connected to the green LED LD4. Active Low.
PA13	PA13 is connected to the red LED LD6. Active Low.
PH7	PH7 is connected to the orange LED LD7. Active High.
PD11	PD11 is connected to the blue LED LD8. Active High.

Figura 3.2: Conexiones de los leds extraídas de la hoja de datos [6]

También se obtiene de la hoja de datos de la placa de desarrollo, los registros a configurar para poder encender y apagar correctamente los LEDs. Cabe destacar que en el Anexo B se ha recopilado la información de estos registros. Recopilando un poco, los pasos a nivel de registro serían los siguientes:

- Configurar el GPIOx como salida de propósito general, escribiendo un 0b01 en los bits correspondientes del GPIO del registro GPIOx_MODER.
- Configurar el GPIOx con PushPull escribiendo un 0b0 en el bit correspondiente del registro GPIOx_OTYPER.
- Configurar el GPIOx con resistencia interna de PullUp escribiendo un 0b01 en el bit correspondiente del registro GPIOx_PUPDR.
- Encender y apagar un LED o bien cambiar el estado de un GPIO, escribiendo en el bit correspondiente del registro GPIOx_BSSR.

3.5.1. Implementación como un driver de plataforma

A continuación se implementará un driver para poder controlar el encendido y apagado de los LEDs rojo, azul y verde. Para esto se va a desarrollar un *platform driver*, explicado en el apartado 2.3.1.

En primer lugar, para crear un *platform driver* son necesarias las secciones siguientes:

- Crear una lista de dispositivos soportados por el driver. Esto se hace creando una lista de estructuras del tipo `of_device_id` e inicializándolas con el nombre que se usará para enlazar el dispositivo con el driver en el campo *compatible*.
- Crear una estructura del tipo `platform_driver` que define al driver y que se registrará en el bus de plataforma.
- Crear la función `probe()` que será llamada cuando se encuentre un dispositivo compatible con el driver. Esta función tiene un rol similar a la función `init()` que se define en los drivers anteriores.
- Crear la función `remove()` que será llamada cuando se descargue el driver. Esta función tiene un rol similar a la función `exit()` que se define en los drivers anteriores.
- Registrar el driver de plataforma en el bus de plataforma, haciendo uso de la función `module_platform_driver()` de la API del kernel.
- Modificar el *device tree* para incluir el dispositivo que se desea controlar.

3.5.1.1. Desarrollo del driver

En el siguiente Listing 3.7 se observa la implementación de los pasos descritos en el subapartado anterior. Ahora, al instanciar un dispositivo en el *device tree* con el campo `compatible = "essovi,RGBleds"`, se sabrá que el driver que lo controla es este.

Listing 3.7: RGBled.c

```
[...]
#include <linux/module.h>
#include <linux/platform_device.h>


static int led_probe(struct platform_device *pdev)
{
    [...}
}

static int led_remove(struct platform_device *pdev)
{
    [...]
}

static const struct of_device_id my_of_ids[] = {

  { .compatible = "essovi,RGBleds"},
```

```
  {},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver led_platform_driver = {
  .probe = led_probe,
  .remove = led_remove,
  .driver = {
    .name = "RGBleds",
    .of_match_table = my_of_ids,
    .owner = THIS_MODULE,
  }
};

module_platform_driver(led_platform_driver);
[...]
```

Continuando con la implementación, en la inicialización del driver va a ser necesario obtener la información del fichero de descripción del hardware, más concretamente la obtención de los registros de los GPIOs a controlar desde el *device tree* y la obtención de sus correspondientes relojes (ya que es necesario habilitarlos y deshabilitarlos mientras se están configurando los registros). También se van a tener que inicializar y crear diferentes estructuras. A continuación se agrupan las funciones a realizar.

1. **Alojar espacio** para la estructura interna a crear haciendo uso de la función de la API del kernel `devm_kzalloc()`.
2. **Obtener el reloj** del banco de GPIOs correspondiente a través del *device tree* haciendo uso de la función de la API del kernel `devm_clk_get()`.
3. **Preparar y habilitar el reloj** de los GPIOs.
4. **Mapear las direcciones** de cada registro sobre el que se desea interactuar.
5. **Configurar los registros** correspondientes para inicializar los GPIOs como salidas con PushPull y resistencia interna de PullUp.
6. **Alojar espacio** para la estructura del dispositivo a crear haciendo uso de la función de la API del kernel `devm_kzalloc()`.
7. **Inicializar la estructura del dispositivo** a crear (del tipo `led_device`, estructura propia que se verá a continuación).
8. **Registrar el dispositivo el kernel**. Este dispositivo estará accesible a través del *sysfs* bajo el directorio `/sys/class/misc/`, además de crearse un nodo para el mismo en el directorio `/dev`.
9. Guardar la información del driver para poder obtenerla a posterior.

Estructuras usadas

Como se desea realizar un driver que pueda controlar los LEDs rojo, verde y azul, se van a definir dos estructuras, una estructura `led_device` que define a cada uno de los

dispositivos LED y que contiene información específica del dispositivo; y una estructura global interna que contiene punteros a los relojes que controlan cada uno de los bancos de los GPIOs a los que pertenecen, y un puntero a cada una de las estructuras de los dispositivos `led_device`.

Listing 3.8: ledRGB structs

```
[...]
// Declare a private structure that will hold each led specific info.
struct led_device
{
  struct miscdevice led_misc_device;
  u32 led_mask_set;
  u32 led_mask_clear;
  const char *led_name;
  char led_value[8];
  struct leds_device *private; // pointer to the global private struct.
};


// Declare a global private structure that will hold common information to all
// the led devices.
struct leds_device
{
  u32 num_leds;
  struct clk *clk_gpioa;
  struct clk *clk_gpiod;
  struct led_device *leds[]; //pointers to each led device private struct.
};
[...]
```

Creación del *device-tree* asociado

A continuación se muestra la instanciación del dispositivo creado en el fichero de descripción del hardware o *device tree*. Cabe destacar que el driver se ha desarrollado para poder controlar los LEDs rojo, azul y verde, pero si no se desea instanciar los tres LEDs, basta con no incluir su nodo correspondiente (nodos red, green y blue) en la configuración. De esta manera, si un nodo no está no se configurará su LED correspondiente ni se creará su dispositivo ni nodo del mismo.

Listing 3.9: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
   ledRGB {
   compatible = "essovi,RGBleds";
   clocks = <&rcc GPIOA>,
            <&rcc GPIOD>;
   clock-names = "GPIOA", "GPIOD";

   red {
          label = "ledred";
   };

   green {
          label = "ledgreen";
   };
```

```
    blue {
17          label = "ledblue";
    };
  };
[...]
```

Implementación del driver

Haciendo uso de los pasos y estructuras descritas se ha implementado el driver cuyo código completo se encuentra en el Listing A.4 del Anexo A. Uno de los aspectos a resaltar es el hecho de que el driver se ha desarrollado para poder controlar los LEDs rojo, verde y azul, creando un dispositivo para cada uno de los LEDs y por tanto controlándose de manera independiente. Además, como se ha comentado, a través del fichero de configuración se seleccionarán aquellos LEDs a controlar incluyendo su nodo correspondiente en el dispositivo. También es importante destacar que la obtención de los datos del fichero de configuración desde el driver se realiza de manera diferente dependiendo de como se estructure el dispositivo en este fichero.

Otros aspectos a destacar son el hecho de que en el driver realizado las operaciones de escritura se realizan indicando el estado del LED a establecer, en formato de texto (*on* o *off*). De la misma manera, al hacer una operación de lectura sobre el dispositivo, se devolverán uno de estos dos estados, según el estado físico del LED.

3.5.2. Implementación como un driver de plataforma UIO

El *Userspace I/O* (UIO) driver es un tipo de driver que Linux incluye (si se habilita en la configuración del kernel) y que permite el acceso a un dispositivo desde el espacio de usuario. Este tipo de driver puede usarse si se desea gestionar alguna interrupción y/o acceder al espacio de memoria del dispositivo (sin usar muchos más recursos del kernel) pero no se desea desarrollar un driver de kernel específico para ello. Su uso simplifica el desarrollo y reduce el riesgo de posibles errores en el sistema operativo. Existen una serie de requisitos para poder desarrollar un driver de este tipo, como que el dispositivo a controlar tenga memoria mapeada, y que pueda ser controlado únicamente escribiendo en esta.

GPIO UIO driver

Aunque generalmente no se implementaría un driver para un GPIO haciendo uso del espacio de usuario UIO, como se desea explorar este tipo de driver, se va a desarrollar un driver de GPIO con este método. Por tanto, el driver de kernel tendrá acceso a la región de memoria de los GPIOs, y el espacio de usuario podrá acceder a la misma a través del driver.

En primer lugar, se va realizar el driver a nivel de kernel. Para su implementación se va hacer uso de la interfaz *UIO* de Linux. En el siguiente Listing 3.10 se observa parte de la API del kernel para gestionar *UIO devices*. Se puede apreciar la propia estructura que define a un dispositivo de esta clase, otra estructura que define su información, así como dos funciones para registrar y dar de baja a estos dispositivos.

Listing 3.10: linux/uio_driver.h

```
struct uio_device {
  struct module *owner;
  struct device  dev;
  int minor;
  atomic_t event;
  struct fasync_struct *async_queue;
  wait_queue_head_t wait;
  struct uio_info *info;
  struct mutex  info_lock;
  struct kobject *map_dir;
  struct kobject *portio_dir;
};

struct uio_info {
  struct uio_device *uio_dev;
  const char  *name;
  const char  *version;
  struct uio_mem  mem[MAX_UIO_MAPS];
  struct uio_port  port[MAX_UIO_PORT_REGIONS];
  long   irq;
  unsigned long  irq_flags;
  void   *priv;
  irqreturn_t (*handler)(int irq, struct uio_info *dev_info);
  int (*mmap)(struct uio_info *info, struct vm_area_struct *vma);
  int (*open)(struct uio_info *info, struct inode *inode);
  int (*release)(struct uio_info *info, struct inode *inode);
  int (*irqcontrol)(struct uio_info *info, s32 irq_on);
};

int uio_register_device(struct module *owner, struct device *parent,
    struct uio_info *info);
void uio_unregister_device(struct uio_info *info);
```

En la implementación del driver de kernel UIO únicamente se tendrá que mapear la memoria que controla al GPIO que se pretende controlar y se habrá de guardar su puntero. Esto se hace con el objetivo de que el driver que se implemente a nivel de usuario, al mapear la memoria del dispositivo, pueda acceder a la memoria de los GPIOs haciendo uso del puntero guardado. Cabe recordar que la gestión de la memoria en Linux se explica en la sección 2.4. A continuación se describen los pasos a realizar:

1. **Obtener la memoria** que contiene los registros de los GPIOs del *device tree* haciendo uso de la función de la API del kernel `platform_get_resource()`.
2. **Obtener el reloj** del banco de GPIOs correspondiente a través del *device tree* haciendo uso de la función de la API del kernel `devm_clk_get()`.
3. **Preparar y habilitar el reloj** de los GPIOs.
4. **Mapear esa memoria** haciendo uso de la función de la API del kernel `devm_ioremap()`.
5. **Inicializar la estructura del dispositivo** a crear (del tipo `uio_device`).

6. **Registrar el dispositivo UIO en el kernel**. Este dispositivo estará accesible a través del *sysfs* bajo el directorio `/sys/class/uio/`. Además se creará un nodo para el mismo en el directorio `/dev`. Todo lo anterior se encuentra garantizado por la función `uio_register_device()`.

Con estos pasos descritos se implementa el driver UIO, su código detallado se adjunta en el Listing A.6 del Anexo A. Es importante apuntar que en la estructura `uio_info` creada no se han asignado las operaciones del driver ya que no se desean reimplementar, por tanto se usarán las operaciones de apertura, cierre y mapeo de memoria por defecto definidas en el *framework* `uio`.

Finalmente, cabe destacar que al tratarse de un *platform driver* será necesario añadir la siguiente instancia del dispositivo físico en el *device tree*.

Listing 3.11: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
   ledUIO {
       compatible = "essovi,UIO";
       reg = <0x50002000 0x1000>;
       clocks = <&rcc GPIOA>;
       clock-names = "GPIOA";
   }
[...]
```

GPIO UIO driver (espacio de usuario)

Tras crear un driver UIO de kernel que permite acceder al espacio de memoria de su dispositivo, se ha de realizar una aplicación a nivel de usuario que interactúe con esa memoria para controlar lo que desea, en este caso, un GPIO. La aplicación que haga uso este driver deberá realizar como mínimo los siguientes pasos.

1. **Apertura del nodo de dispositivo** mediante la llamada al sistema `open()`.
2. **Mapeado de la memoria del dispositivo** mediante la llamada `mmap()`.
3. Acceso a la memoria con los permisos que se hayan pedido (en este caso interesa tener permisos de lectura y escritura sobre la memoria).
4. **Liberación de la memoria mapeada** mediante la llamada `munmap()`.
5. **Cierre del nodo de dispositivo** mediante la llamada al sistema `close()`.

En el Listing 3.12 se muestra el esqueleto de la aplicación desarrollada para complementar al driver UIO del kernel y poder controlar los LEDs. El código completo de la aplicación se encuentra en el Listing A.8 del Anexo A. En este extracto únicamente se muestran las funciones relativas a los pasos mencionados en la lista anterior, es decir las correspondientes a las operaciones de apertura y mapeado de memoria del dispositivo, así como sus complementarias de liberación de la memoria y cierre del dispositivo.

Listing 3.12: uio_app.c

```
[...]

  int devUIO_fd;
  const unsigned int uio_size;
  void *driver_mem;
  int ret;

  // Open file descriptor.
  devUIO_fd = open("/dev/uio0", O_RDWR | O_SYNC);
  if(devUIO_fd < 0)
  {
    printf("Failed opening the device \n");
    return -1;
  }

  // Get the size to map by reading file /sys/class/uio/uio0/maps/map0/size
  [..]

  // Do the uio_size mapping.
  driver_mem = mmap(NULL, uio_size, PROT_READ | PROT_WRITE, MAP_SHARED, devUIO_fd, 0);
  if(driver_mem == MAP_FAILED)
  {
    printf("Invalid devUIO mmap");
    close(devUIO_fd);
    return -1;
  }

[...]
    // Unmap memory mapped.
    ret = munmap(driver_mem, uio_size);
    if(ret < 0 )
    {
    printf("Error unmaping\n");
    close(devUIO_fd);
    return -1;
  }

    // Close device.
  close(devUIO_fd);

[...]
```

3.6. Driver LCD (LCM1602)

A continuación se va a explicar el desarrollo del driver de kernel realizado para una pantalla LCD de 16x2, que corresponde al modelo LCM1602.

En primer lugar cabe destacar que la pantalla se va a conectar a la placa de desarrollo STM32MP157C-DK2 a través de los pines GPIO, en concreto se va a usar el módulo de expansión de GPIO del que dispone la placa de STMicroelectronics. En la Figura 3.3 se pueden observar las conexiones del LCD.

Esta pantalla LCD tiene diversas funcionalidades y configuraciones, las cuales se in-

Pin no.	Symbol	External connection	Function
1	Vss	Power supply	Signal ground for LCM
2	V_{DD}		Power supply for logic for LCM
3	V_0		Contrast adjust
4	RS	MPU	Register select signal
5	R/W	MPU	Read/write select signal
6	E	MPU	Operation (data read/write) enable signal
7~10	DB0~DB3	MPU	Four low order bi-directional three-state data bus lines. Used for data transfer between the MPU and the LCM. These four are not used during 4-bit operation.
11~14	DB4~DB7	MPU	Four high order bi-directional three-state data bus lines. Used for data transfer between the MPU
15	LED+	LED BKL power supply	Power supply for BKL
16	LED-		Power supply for BKL

Figura 3.3: Imagen extraída de la hoja de datos del componente LCM1602 [7]

cluyen todas en su hoja de datos [7]. A continuación se describen las que se implementarán en el driver a realizar.

Funcionalidades que proveerá el driver:

- **Modo de operación de 4 bits**, por lo que los pines DB0-DB3 no se usarán
- **Elección** de la configuración del LCD del **número de líneas** a usar 1 ó 2, mediante el ***device tree***
- **Elección de los pines** de la placa STM32MP157 a usar, mediante el ***device tree***
- **Escritura** de mensajes en el LCD
- **Movimiento del cursor** a una posición concreta
- **Borrado de la pantalla**

3.6.1. Funcionamiento del LCD

Para conocer las instrucciones de la pantalla a usar, es necesario obtener esta información de su hoja de datos [7].

Como aspectos a destacar, la pantalla tiene un proceso de inicialización en el que se configura, entre muchas otras cosas, el modo de operación, el número de líneas a usar, si se desea cursor visible, si se desea que el cursor parpadee y/o se desplace. Esta inicialización la tendrá que realizar al inicio el driver a implementar.

También se especifican las instrucciones para limpiar la pantalla, mover el cursor a una posición específica y escribir un carácter.

3.6.2. Desarrollo del driver

El driver a desarrollar será del tipo driver de plataforma, permitiendo configurar la información física del LCD en el *device tree*. Para realizar la escritura y lectura de los GPIOs, en lugar de realizarse a tan bajo nivel como en el ejemplo de la sección 3.5, se va a hacer uso de la interfaz de consumo de los descriptores GPIO que provee Linux. Esta interfaz identifica cada GPIO como una estructura del tipo `gpio_desc`. Además, todas las funciones pertenecientes a esta interfaz contienen el prefijo *gpiod_*.

3.6.2.1. Estructuras usadas

Además de la interfaz de los descriptores GPIO, para la implementación del driver se va a hacer uso de estructuras propias. Estas estructuras almacenan información relativa al dispositivo y al driver que se pretenden crear. En el Listing 3.13 se observan las siguientes estructuras.

En primer lugar, se encuentra la estructura **charlcd**, que corresponde con una estructura privada que almacena la información del driver del LCD. Esta contiene diferentes variables con información física de la pantalla, así como un puntero a la estructura que define las operaciones que realiza el driver (`file_operations`), una instancia del dispositivo (del tipo `misc_device`) , y un array con los punteros a los descriptores GPIO que usará el driver del LCD.

En segundo lugar, se observa la estructura **file_operations** de las operaciones que realizará el driver. En este caso el driver únicamente permitirá realizar escrituras (sin tener en cuenta las operaciones siempre necesarias de apertura y cierre).

Listing 3.13: display.c

```
[...]
/// Declare a private structure that will hold lcd specific info.
struct charlcd {
  const struct file_operations *ops;
  struct miscdevice charlcd_dev;
  int ifwidth;
  int height;
  int width;
  struct gpio_desc *pins[PIN_NUM];
};

/// Define the structure with the supported operations of the device.
static const struct file_operations charlcd_fops = {
  .write = charlcd_write,
  .open = charlcd_open,
  .release = charlcd_release,
};

[...]
```

3.6.3. Creación del *device-tree* asociado

En el Listing 3.14 se observa el trozo a instanciar en el *device tree* para que al cargar el módulo en el kernel, se pueda realizar el proceso de *binding* anteriormente descrito en el apartado 2.3.1.1.

Como se observa, se incluyen los pines asociados a los datos (en orden, siendo el primer GPIO descrito el equivalente al pin DB4 al usar el modo de 4 bits), el pin asociado al 'enable' (pin E en la documentación de la Figura 3.3) y al 'rs' (o RS). También se incluyen dos propiedades que indican la anchura y altura de la pantalla LCD.

Listing 3.14: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
   lcd-display {
       compatible = "lcd,LCM1602";
       data-gpios = <&gpiof 8 GPIO_ACTIVE_HIGH>,
          <&gpiof 9 GPIO_ACTIVE_HIGH>,
          <&gpioa 8 GPIO_ACTIVE_HIGH>,
          <&gpiob 10 GPIO_ACTIVE_HIGH>;

       enable-gpios = <&gpiob 12 GPIO_ACTIVE_HIGH>;
       rs-gpios = <&gpiof 7 GPIO_ACTIVE_HIGH>;

       display-width-chars = <16>;
       display-height-chars = <2>;
 };
[...]
```

3.6.3.1. Obtención de la información del dispositivo físico del *device tree*

Para poder obtener la información del fichero de descripción del hardware se van a hacer uso de funciones de la interfaz de los descriptores GPIO (para la información de los pines GPIO), además de otras funciones para obtener otros valores numéricos. En concreto se van a usar las siguientes:

- `gpiod_count()`:

  Se hace uso de esta función para obtener el número de GPIOs asociados con un dispositivo y función. En este caso, se va a usar para obtener el número de GPIOs del tipo 'data-gpios' del dispositivo instanciado. Esto se hace por si se quiere a futuro dar soporte para que el driver use el modo de 8 bits, en el que intervendrían 8 GPIOs.

- `devm_gpiod_get_index()`:

  Esta función se usa para obtener el descriptor GPIO (asociado con un dispositivo y función) dado un índice. Se va a usar para obtener cada uno de los descriptores GPIO del tipo 'data-gpios', haciendo un bucle hasta el número de descriptores obtenidos con la función anterior. Esta función también permite establecer unos *flags* que pueden servir para inicializar el GPIO a un valor en concreto.

- `devm_gpiod_get()`:

  Esta función se usa para obtener un único descriptor GPIO (asociado con un dispositivo y función). Se va a usar para obtener los descriptores GPIO del tipo 'enable-gpios' y 'rs-gpios', ya que cada uno de estos tipos únicamente tiene un descriptor. También permite establecer unos *flags*.

- `device_property_read_u32()`:

  Esta función se usa para obtener una propiedad numérica sin signo (asociada al dispositivo) del fichero de descripción del hardware. Se usará para obtener los parámetros de configuración de ancho y alto de la pantalla LCD.

El uso de todas estas funciones mencionadas, se puede observar en el Listing A.10 del Anexo A.

3.6.4. Definición de las funciones implementadas

Para la implementación de las funcionalidades descritas se han definido diferentes funciones a distintos niveles de abstracción, con el objetivo de reutilizar el mayor código posible.

A continuación se observa la declaración de cada una de estas funciones. El nivel más alto correspondería a las propias funciones del driver, las que definen las operaciones que realiza (apertura, cierre y escritura para esta implementación). Se puede apreciar a posterior un segundo nivel en el que se declaran funciones que implementan las funcionalidades descritas (inicialización, borrado y escritura en la pantalla y movimiento del cursor). Finalmente se observa un último nivel de abstracción con funciones auxiliares.

Cabe recordar que el código completo del driver se encuentra en el Listing A.10 del Anexo A.

Listing 3.15: display.c

```
[...]
////////////////////// Function definitions ////////////////////////////
// Driver functions definitions.
static int charlcd_open(struct inode *inode, struct file *file);
static int charlcd_release(struct inode *inode, struct file *file);
static int charlcd_write(struct file *file, const char __user *buf,
                         size_t count, loff_t *ppos);

// Internal high-level functions.
static void lcd_init(struct charlcd *lcd);
static void clear(struct charlcd *lcd);
static int move_cursor(struct charlcd *lcd, unsigned x, unsigned y);
static void print_msg(struct charlcd *lcd, char const *msg);

// Internal low-level functions.
static void write_4bitsmode(struct gpio_desc *pins[], uint8_t command,
                            unsigned rs_value);
static void print_char(struct charlcd *lcd, char c);
static void clockPulse(struct gpio_desc *pins[]);
[...]
```

3.7. Driver GPIO con interrupciones

En numerosas aplicaciones en sistemas embebidos resulta interesante poder controlar interrupciones. Como se ha comentado en la sección 2.2, los drivers a nivel de kernel permiten el manejo de interrupciones, por lo que se va a explorar esta funcionalidad en la siguiente implementación.

El tema de las interrupciones en Linux es muy extenso y complejo, abarcando un sinfín de posibilidades para gestionar todo tipo de interrupciones hardware y software. El desarrollo del ejemplo a realizar se va a enfocar en el manejo de una sencilla interrupción hardware, en la que el sistema operativo llamará al manejador de la interrupción generada que ejecutará la rutina deseada.

En concreto, se va a realizar una sencilla aplicación con un botón y un LED conectados a los GPIOs de la placa STM32MP157C-DK2. Cabe destacar que inicialmente se iba a hacer uso de los botones de usuario y de los LEDs incorporados en la misma placa, pero al revisar las conexiones en su hoja de datos, dos de los LEDS comparten pines con los dos botones de usuario, lo que imposibilita su uso simultáneo. Por tanto, se va a hacer uso de un botón conectado a uno de los pines del expansor GPIO (configurable por medio del *device tree*), y de uno de los LEDs de la placa, en concreto el LED naranja, aunque se puede cambiar el GPIO a usar también desde el fichero de configuración del hardware.

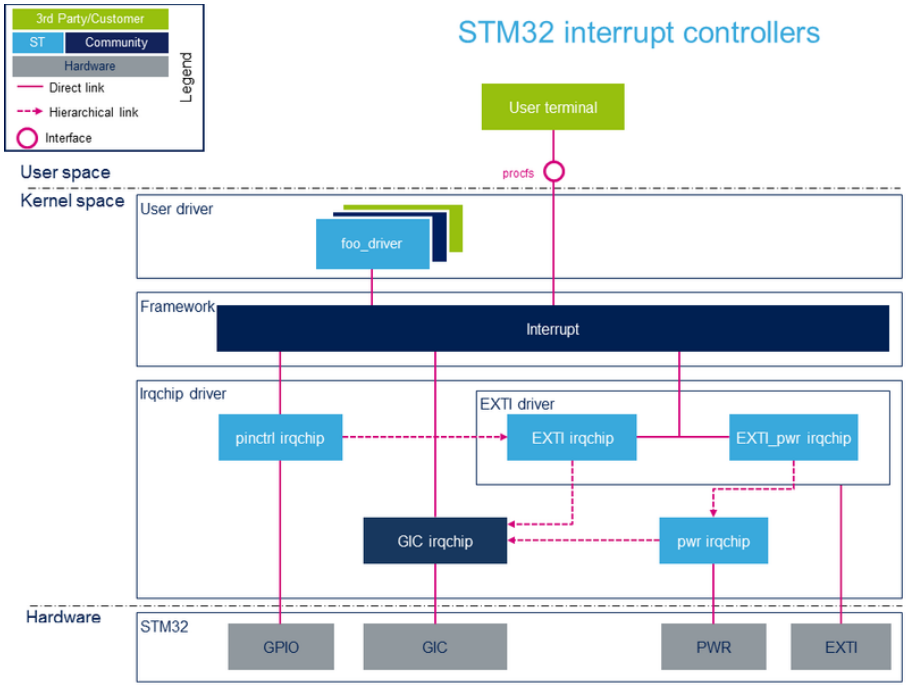


Figura 3.4: Esquema de las interfaces que intervienen en las interrupciones. Extraída de [8]

3.7.1. Interrupciones en Linux

La capa del kernel de Linux que maneja las interrupciones se encuentra dividida en dos partes: una parte genérica y otra relacionada con los pines hardware, relativa al *irqchip*.

La parte genérica provee una API para solicitar y configurar una línea de interrupción, así como crea identificadores únicos para cada interrupción realizando mapeados virtuales. La parte del *irqchip* se encarga de manejar los accesos al hardware y otros aspectos específicos.

En la Figura 3.4 se observa el esquema obtenido del fabricante en relación a todas las interfaces que intervienen en el sistema de interrupciones.

Para el ejemplo a desarrollar, como se desea usar un GPIO como interrupción, aplica la parte relativa al *pinctrl irqchip*. También se observa que el fabricante ofrece este soporte para el manejo de las interrupciones GPIO, activas por defecto en la configuración del sistema. Por tanto, será necesario saber como configurar el *device tree* para hacer que el dispositivo a crear pueda manejar una interrupción.

3.7.2. Configuración del *device tree* para manejo de interrupciones

En el fichero de descripción del hardware se incluyen todos los dispositivos que conforman al sistema. Para la configuración de interrupciones existe una distinción según la naturaleza del dispositivo.

- **Clientes de interrupciones** : corresponden a los dispositivos que generan interrupciones. Tienen un controlador de interrupciones asociado.
- **Controladores de interrupciones** : corresponden a los dispositivos que controlan las interrupciones.

La configuración del sistema proporcionada por el fabricante incluyen diferentes dispositivos (por ejemplo, *gpioa*) que agrupan los diferentes GPIOs de un mismo banco y resultan ser controladores de GPIO (*gpio-controller*) y de interrupciones (*interrupt-controller*). Esto se puede observar en el trozo de *device tree* extraído en el Listing 3.16.

Por tanto, como ya existe un dispositivo que realiza el control de la interrupción, únicamente se habrá de configurar el dispositivo que se cree para generar la interrupción e indicar cuál es su controlador de interrupción correspondiente.

Listing 3.16: arch/arm/boot/dts/stm32mp151.dtsi

```
[...]
 pinctrl: pin-controller@50002000 {
   #address-cells = <1>;
   #size-cells = <1>;
   compatible = "st,stm32mp157-pinctrl";
   ranges = <0 0x50002000 0xa400>;
   interrupt-parent = <&exti>;
   st,syscfg = <&exti 0x60 0xff>;
```

```
    hwlocks = <&hsem 0 1>;
    pins-are-numbered;

        [...]

    gpiod: gpio@50005000 {
      gpio-controller;
      #gpio-cells = <2>;
      interrupt-controller;
      #interrupt-cells = <2>;
      reg = <0x3000 0x400>;
      clocks = <&rcc GPIOD>;
      st,bank-name = "GPIOD";
      };
[...]
```

3.7.3. Desarrollo e implementación del driver

3.7.3.1. Creación del *device-tree* asociado

En el Listing 3.17 se observa el trozo a instanciar en el *device tree* para que al cargar el módulo en el kernel, se pueda realizar el proceso de *binding* anteriormente descrito.

Como se puede apreciar, se incluye el pin que generará la interrupción (pin PD13). Para esto es necesario pasar el descriptor GPIO al dispositivo (por medio del campo `gpio`), así como especificar la lista de interrupciones del dispositivo, haciendo uso de la propiedad `interrupts`. Otro aspecto importante ya mencionado, es el hecho de indicar el controlador de las interrupciones asociado (por medio del campo `interrupt-parent`), que en este caso corresponde con el propio banco GPIO.

También se indica en la descripción del dispositivo, el descriptor GPIO del LED que se usará para cambiar de estado cada vez que se genere una interrupción, que corresponde con el pin PH7.

Listing 3.17: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
    irq-key {
        compatible = "essovi,irqGPIO";
          label = "PB_USER";
          gpios = <&gpiod 13 GPIO_ACTIVE_HIGH>;
          led-gpios = <&gpioh 7 GPIO_ACTIVE_HIGH>;
          interrupt-parent = <&gpiod>;
          interrupts = <13 IRQ_TYPE_EDGE_RISING>;
  };
[...]
```

3.7.3.2. Estructuras y funciones usadas

A nivel de desarrollo, para crear el driver se van a hacer uso de diferentes funciones de la API del kernel. A continuación en el Listing 3.19 se incluye la definición de las funciones relativas al tratamiento de las interrupciones.

Listing 3.18: include/linux/interrupt.h

```
[...]

// Funciones de la API del kernel relativas a las interrupciones.
int request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long flags, const char *name, void *dev);

const void *free_irq(unsigned int irq, void *dev);

int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
    unsigned long irqflags, const char *devname, void *dev_id);

void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);

// Prototipo de la función del manejador de interrupciones.
irqreturn_t (*irq_handler_t)(int irq_nb, void *dev_id);

[...]
```

Entrando un poco más en detalle, las funciones `request_irq()` y `devm_request_irq()` alojan los recursos de la interrupción y habilitan la línea de interrupción y el manejo de la misma. Entre sus argumentos, destacan el identificador IRQ que el kernel le ha adjudicado, la función que realizará el manejo de la interrupción y otros flags que definen el comportamiento de la interrupción. También destacar el último argumento que corresponde con un puntero que puede tomar cualquier valor, por lo que resulta útil para pasar al manejador información interna del driver del dispositivo. La diferencia entre ambas funciones radica en el hecho de que `devm_request_irq()` está asociada al dispositivo que lo usa, lo que permite liberar automáticamente todos los recursos que ha alojado cuando el dispositivo se elimina. Debido a esta gran ventaja, que simplifica el uso de las interrupciones al no tener que controlar la liberación de los recursos, esta función va a ser la escogida en el desarrollo del driver.

Con respecto a las funciones `free_irq()` y `devm_free_irq()`, estas sirven para liberar todos los recursos obtenidos con las funciones anteriormente descritas. La diferencia entre ambas radica en que la función `devm_free_irq()` se encuentra asociada al dispositivo que la usa.

Lo último que se observa en el Listing 3.19 corresponde con el prototipo de la función del manejador de interrupciones, por lo que todos los manejadores que se escriban tendrán que seguir este formato de función.

Estructuras usadas

Con respecto a las estructuras usadas en el driver a implementar, únicamente se va a hacer uso de una estructura interna que permita almacenar el dispositivo a crear y un puntero al descriptor GPIO del LED, con el objetivo de encapsular esta información y poder pasársela al manejador de la interrupción para que pueda hacer uso de la misma. La estructura es la siguiente.

Listing 3.19: include/linux/interrupt.h

```
[...]
// Estructura interna del driver.
static struct irq_dev
```

```
{
  struct gpio_desc *led_gpio_ntfy;
  struct miscdevice dev;
};
[...]
```

3.7.3.3. Inicialización del driver

El driver a desarrollar será del tipo *platform driver*, por lo que su estructura será muy similar a los drivers de este tipo ya realizados. Con respecto a las operaciones que realizará este driver, únicamente implementará las de apertura y cierre por defecto, ya que el objetivo del driver es el del manejo de una interrupción.

A continuación se van a describir los pasos a realizar en la inicialización del driver, entre los que se incluyen configuraciones específicas para las interrupciones.

1. **Alojar espacio** para la estructura interna a crear haciendo uso de la función de la API del kernel `devm_kzalloc()`.
2. **Obtener los GPIOs** correspondiente a través del *device tree* haciendo uso de la función de la API del kernel `devm_gpiod_get()`.
3. **Configurar los GPIOS**. Configurar el GPIO correspondiente al LED como salida, y el GPIO usado para la interrupción como entrada.
4. **Obtener un identificador IRQ** del kernel haciendo uso de la función de la API del kernel `gpiod_to_irq()`.
5. **Alojar la interrupción** en el kernel, haciendo uso de la función de la API del kernel `devm_request_irq()`.
6. **Inicializar la estructura del dispositivo** a crear (del tipo `miscdevice`).
7. **Registrar el dispositivo el kernel**. Este dispositivo estará accesible a través del *sysfs* bajo el directorio `/sys/class/misc/`, además de crearse un nodo para el mismo en el directorio `/dev`.
8. Guardar la información del driver para poder obtenerla a posterior.

Siguiendo todos los pasos descritos así como haciendo uso de las funciones y estructuras explicadas, se puede realizar el driver cuyo código se incluye en el A.14 del Anexo A.

3.8. Driver cliente I2C (Nunchuck)

En la siguiente sección se explica el desarrollo del driver de kernel realizado para interactuar con el mando *Wii Nunchuk* [9].

El mando *Nunchuk* usa el protocolo I2C para comunicarse, disponiendo de la distribución de pines que se observa en la Figura 3.5. También es conocida su dirección I2C, que corresponde con la 0x52, así como la frecuencia de comunicación que emplea, 100KHz. El mando contiene un acelerómetro del que se puede obtener su información en los tres ejes, así como un *joystick* y dos botones. Todos estos datos están accesibles en sus respectivos registros, adjuntos en el Anexo C.

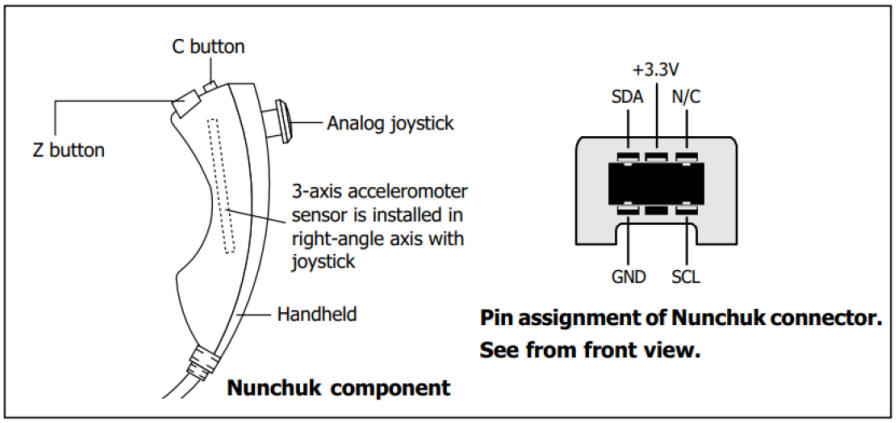


Figura 3.5: Distribución de pines del mando Nunchuk

Para conectar el mando *Nunchuk* a la placa STM32MP157-DK2, se ha de buscar una interfaz de tipo I2C que no se encuentre en uso. Repasando su hoja de datos se obtiene que la placa de desarrollo usada tiene 3 interfaces de este tipo: I2C1, I2C4 e I2C5, de las cuales únicamente la última interfaz se encuentra libre en la configuración dada por el fabricante. Por tanto, se hace uso de la interfaz I2C5 en la que se conectan los pines SCL y SDA del mando a los pines PA11 y PA12, respectivamente, de la placa STM32MP157C-DK2 (de acuerdo a la tabla de la Figura 3.6).

Connector	Pin name	Signal name	STM32 pin	Comment
CN13	1	ARD_D8	PG3	IO
	2	ARD_D9	PH6	TIM12_CH1
	3	ARD_D10	PE11	SPI4_NSS and TIM1_CH2
	4	ARD_D11	PE14	SPI4_MOSI and TIM1_CH4
	5	ARD_D12	PE13	SPI4_MISO
	6	ARD_D13	PE12	SPI4_SCK
	7	GND	-	GND
	8	VREFP	-	VREF+
	9	ARD_D14	PA12	I2C5_SDA
	10	ARD_D15	PA11	I2C5_SCL

Figura 3.6: Conexiones de la placa STM32MP157C-DK2 relativas al conector de Arduino

3.8.1. I2C Device Model

Para poder realizar un driver de un dispositivo I2C cliente, es necesario conocer como maneja Linux este protocolo. Para el mismo, Linux posee un subsistema propio basado en su modelo de dispositivo y formado por diferentes controladores, cada uno a un nivel de abstracción diferente.

El primer nivel del subsistema I2C, es el núcleo del bus I2C o *I2C bus core*, que provee la interfaz de soporte entre el driver de un cliente individual y numerosos buses I2C maestros. Maneja el arbitraje del bus, los reintentos y otros detalles del protocolo I2C.

El siguiente nivel corresponde con los drivers del controlador I2C o *I2C controller drivers*, que contienen funciones específicas para leer y escribir sobre las direcciones hardware del controlador I2C.

El último nivel corresponde con los drivers de dispositivos I2C o *I2C device drivers*, que implementan el control de los dispositivos I2C. Cada driver depende directamente del dispositivo, y hace uso de la API del núcleo I2C para enviar y recibir datos del dispositivo I2C.

3.8.2. Desarrollo e implementación del driver como un driver I2C

Con todo lo anterior relativo al modelo de dispositivo del subsistema I2C, se puede deducir que el driver a desarrollar será del último tipo, perteneciente a un driver de dispositivo I2C.

La estructura del driver será similar a los realizados anteriormente con la diferencia de que, en lugar de crear un driver de plataforma, se creará un controlador de I2C, del tipo `i2c_driver`. A continuación en el Listing 3.20 se muestran algunas de las funciones y estructuras de la API del kernel para I2C.

Listing 3.20: linux/i2c.h

```
[...]
struct i2c_driver {
    int (*probe)(struct i2c_client *client, const struct i2c_device_id *id);
    int (*remove)(struct i2c_client *client, const struct i2c_device_id *id);
    int (*probe_new)(struct i2c_client *client);
    void (*shutdown)(struct i2c_client *client);
    void (*alert)(struct i2c_client *client, enum i2c_alert_protocol protocol,
        unsigned int data);
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    struct device_driver driver;
    const struct i2c_device_id *id_table;
    int (*detect)(struct i2c_client *client, struct i2c_board_info *info);
    const unsigned short *address_list;
    struct list_head clients;
};

void i2c_set_clientdata(struct i2c_client *client, void *data);
void* i2c_get_clientdata(const struct i2c_client *client);
int i2c_master_send(const struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(const struct i2c_client *client, char *buf, int count);
```

Para el desarrollo de un *i2c driver* son necesarias las siguientes acciones:

- Crear una lista de dispositivos soportados por el driver, haciendo uso de una lista de estructuras del tipo `of_device_id`, como se ha hecho en anteriores implementaciones.
- Crear una estructura del tipo `i2c_driver` que define al driver de tipo I2C y que se registrará en el bus I2C.
- Crear la función `probe_new` que será llamada cuando se encuentre un dispositivo compatible con el driver.
- Crear la función `remove` que será llamada cuando el driver deje de estar cargado en el kernel.
- Crear las funciones que implementen las operaciones que realizará el driver.
- Registrar el driver I2C en el bus I2C haciendo uso de la función `module_i2c_driver` de la API del kernel.
- Modificar el *device tree* para incluir el dispositivo a controlar, en este caso el *Nunchuk*.

A continuación en el Listing 3.21 se observa parte de la estructura del driver en la que se realizan muchas de las funciones descritas.

Listing 3.21: nunchuk.c

```
[...]
// Definición de los dispositivos compatibles con el driver
static const struct of_device_id nunchuk_of_match[] = {
    { .compatible = "nunchuk"},
    {}
};
MODULE_DEVICE_TABLE(of, nunchuk_of_match)

// Crear la estructura del driver i2c_driver
static struct i2c_driver nunchuk_driver = {
    .driver = {
        .name = "nunchuk",
        .owner = THIS_MODULE,
        .of_match_table = nunchuk_of_match,
    },
    .probe_new = nunchuk_probe,
    .remove = nunchuk_remove,
};

// Registra el driver en el bus I2C
module_i2c_driver(nunchuk_driver);
MODULE_DESCRIPTION("Nunchuk I2C driver")
[...]
```

3.8.2.1. Estructuras usadas

Además de la API I2C, para la implementación del driver se va a hacer uso de estructuras propias que almacenan información relativa al dispositivo y al driver a crear. En el Listing 3.22 se observan las siguientes estructuras.

En primer lugar, se encuentra la estructura **data** que corresponde con una estructura privada que almacena los datos obtenidos del *Wii Nunchuk*. La estructura almacenará la información de los ejes X e Y del joystick, de los 3 ejes del acelerómetro, y de ambos botones.

En segundo lugar, se encuentra la estructura **nunchuk** que es una estructura privada que contiene una instancia del dispositivo a crear (del tipo `miscdevice` como en los anteriores drivers), y un puntero a la estructura que define un cliente I2C, del tipo `i2c_client`.

Finalmente, se observa la estructura `file_operations` de las operaciones que realizará el driver. En este caso el driver únicamente permitirá realizar lecturas, además de las operaciones esenciales de apertura y cierre.

Listing 3.22: nunchuk.c

```
[...]
/// Declarar una estructura privada para guardar los datos del nunchuk.
static struct data {
  uint8_t joy_x;
  uint8_t joy_y;
  uint8_t accel_x;
  uint8_t accel_y;
  uint8_t accel_z;
  bool C_pressed;
  bool Z_pressed;
};

/// Declarar una estructura privada para contener información específica del nunchuk.
struct nunchuk {
  struct miscdevice nunchuk_dev;
  struct i2c_client *client;
};

/// Define las operaciones que realizará el driver.
static const struct file_operations nunchuk_fops = {
    .open = nunchuk_open,
    .release = nunchuk_release,
    .read = nunchuk_read,
};
[...]
```

3.8.2.2. Creación del *device-tree* asociado

En el Listing 3.23 se observa el trozo a instanciar en el *device tree* para que al cargar el módulo en el kernel, se pueda realizar el proceso de *binding* anteriormente descrito en el apartado 2.3.1.1. Cabe destacar que al querer usar la interfaz I2C5, las modificaciones se hacen sobre su nodo.

Con respecto a las modificaciones en las propiedades del I2C5, se ha de habilitar el bus

cambiando la propiedad *status* a `"okay"`, así como cambiar la frecuencia de comunicación para que corresponda con la del Nunchuk. Finalmente, se añade el dispositivo I2C a instanciar, en el que se indica la dirección I2C del *Nunchuk* mediante la propiedad *reg*.

Listing 3.23: arch/arm/boot/dts/stm32mp15xx-dkx.dtsi

```
[...]
&i2c5 {
  pinctrl-names = "default", "sleep";
  pinctrl-0 = <&i2c5_pins_a>;
  pinctrl-1 = <&i2c5_sleep_pins_a>;
  i2c-scl-rising-time-ns = <185>;
  i2c-scl-falling-time-ns = <20>;
  clock-frequency = <100000>;
  /* spare dmas for other usage */
  /delete-property/dmas;
  /delete-property/dma-names;
  status = "okay";

  nunchuk: nunchuk@52 {
    compatible = "nunchuk";
    reg = <0x52>;
  };
};
[...]
```

3.8.2.3. Inicialización del driver

Aunque en apartados anteriores se han descrito los pasos para inicializar los distintos drivers creados, al tratarse de un driver de un tipo diferente, se cree conveniente enumerarlos de nuevo, con el objetivo de remarcar sus diferencias.

1. **Alojar espacio** para la estructura interna a crear haciendo uso de la función de la API del kernel `devm_kzalloc()`.
2. **Asociar el cliente I2C con la estructura interna** para almacenar sus datos y poder acceder posteriormente. Esto se hace con la función `i2c_set_clientdata()`.
3. **Almacenar el dispositivo cliente en la estructura interna**.
4. **Inicializar la estructura del dispositivo** a crear (del tipo `miscdevice`).
5. **Almacenar el dispositivo a crear en la estructura interna**.
6. **Inicializar el *Nunchuk***.
7. **Registrar el dispositivo el kernel**. Este dispositivo estará accesible a través del *sysfs* bajo el directorio `/sys/class/misc/`, además de crearse un nodo para el mismo en el directorio `/dev`.

Todos estos pasos se han seguido para poder realizar la función `probe` del driver creado, cuyo código se observa en el Listing A.16 del Anexo A.

3.8.2.4. Definición de las funciones implementadas

A la hora de implementar el driver se han definido diferentes funciones a distintos niveles de abstracción. A continuación se observa la declaración de cada una de estas funciones. En primer lugar se observan las propias funciones del driver, las que definen las operaciones que este realiza (apertura, cierre y lectura para esta implementación). En el segundo nivel se declaran funciones para inicializar la conexión con el mando y desempaquetar la información leída del mismo. Finalmente se observa un último nivel de abstracción para leer los registros del dispositivo I2C.

Cabe recordar que el código completo del driver se encuentra en el Listing A.16 del Anexo A.

Listing 3.24: nunchuk.c

```
[...]
////////////////////// Function definitions ////////////////////////////

// Driver functions definitions.
static int nunchuk_open(struct inode *inode, struct file *file);
static int nunchuk_release(struct inode *inode, struct file *file);
static ssize_t nunchuk_read(struct file *file, char __user *buf, size_t count,
    loff_t *ppos);

// Internal high-level functions.
static int nunchuk_get_data(struct i2c_client *client, struct data *nunch_data);
static int nunchuk_init(struct i2c_client *client);

// Internal low-level functions.
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf, int buf_size);

[...]
```

Capítulo 4

Presentación de resultados

En el siguiente capítulo se expondrán los resultados de los controladores de los diferentes dispositivos realizados, con el objetivo de demostrar las funcionalidades que poseen cada uno de ellos.

4.1. Conceptos básicos

En esta sección se van a introducir conceptos básicos de Linux que será necesario conocer para poder comprender mejor los resultados de los drivers que se van a presentar.

En primer lugar, como se ha comentado anteriormente, los drivers se implementan como módulos del kernel, que pueden ser cargados y descargados en tiempo de ejecución. Todos los módulos compilados del kernel y listos para ser cargados se encuentran en la ruta `/lib/modules/5.10.61/kernel`, siendo la parte numérica la versión del kernel. Los drivers que se implementen se cargarán en la ruta `/lib/modules/5.10.61/my-modules` para mantenerlos separados del kernel original.

Como se ha visto, al cargar un driver se crea su dispositivo asociado, el cual resulta accesible a través de su nodo de dispositivo, bajo el directorio `/dev`. Este nodo es necesario para que se pueda acceder al dispositivo a través de la capa de usuario. Para esto, lo más común es realizar aplicaciones que hagan uso del nodo de dispositivo (cabe recordar que se trata de un fichero) y que lean, escriban o realicen la operación pertinente sobre el mismo. Una manera muy rápida y sencilla para interactuar desde la capa de usuario con dispositivos de tipo *char* es hacer uso de los comandos de Linux `echo` y `cat`, únicamente útiles en el caso de que el driver del dispositivo permita leer y/o escribir sobre el mismo. El comando `echo` sirve para escribir una serie de caracteres en consola, aunque si se combina con el símbolo `>`, que redirige la salida a otro fichero, se consigue escribir sobre el nodo de dispositivo deseado. El comando `cat`, se puede usar para visualizar el contenido de un fichero, y por tanto, leer de un nodo de dispositivo. En los siguientes apartados se hará uso de estos comandos para interactuar con los dispositivos creados.

4.2. Driver *dummy*

El primer driver desarrollado en la sección 3.4 corresponde con un driver *dummy*, es decir, sin relación real sobre el hardware. Es importante recordar que este driver crea un dispositivo `dummy`. Además este dispositivo pertenece a una clase de dispositivo también creada por el driver, denominada `my_dummy_class`. En la Figura 4.1 se observa la información del módulo del kernel desarrollado, en la que se incluyen diferentes datos del módulo, como nombre, autor, licencia o breve descripción del mismo, entre otros.

```
root@stm32mp1:/lib/modules/5.10.61/my-modules# modinfo dummy_char_driver.ko
filename:       /lib/modules/5.10.61/my-modules/dummy_char_driver.ko
description:    This is a module that interacts with the system call
author:         Esther Soriano
license:        GPL
depends:
name:           dummy_char_driver
vermagic:       5.10.61 SMP preempt mod_unload modversions ARMv7 p2v8
root@stm32mp1:/lib/modules/5.10.61/my-modules#
```

Figura 4.1: Información del módulo de kernel desarrollado.

El driver desarrollado permite que el dispositivo realice operaciones de lectura y escritura, almacenando los mensajes escritos sobre el dispositivo para volcarlos cuando se realice una lectura sobre el mismo. En la Figura 4.2 se observa el *log* del kernel (a la izquierda) y los comandos usados para interactuar con el dispositivo para realizar estas operaciones (a la derecha).

Profundizando un poco más en la Figura 4.2, se observa como inicialmente el nodo del dispositivo `/dev/dummy` no existe, así como tampoco existe la clase del dispositivo en el directorio `/sys/class/my_dummy_class`, pues el módulo no se encuentra cargado en el kernel. Al ejecutar el comando indicado en color naranja, que hace la carga del módulo en el kernel, se observan los mensajes marcados en el mismo color en el *log* del kernel, que indican que se le ha dado al driver el número del identificador mayor del driver (nº 241), así como que se ha creado la clase del dispositivo y el dispositivo. Para comprobar lo anterior, se busca el nodo de dispositivo `/dev/dummy` y la clase del dispositivo en el directorio `/sys/class/my_dummy_class`. Como se observa en la línea de comandos, ambos existen. También se puede apreciar que de toda la información ofrecida en el nodo del dispositivo, aparecen los números 241 y 0, que corresponden a los identificadores mayor y menor del driver, respectivamente. Por último, también se observa el tipo de dispositivo y los permisos que tiene, mediante la codificación `crw-------`, en la que el primer carácter *c* indica que es un dispositivo de tipo *char* o de carácter.

Para comprobar las funcionalidades descritas, se ejecuta en primer lugar el comando resaltado en color amarillo, que realiza una lectura sobre el nodo de dispositivo. Como se observa en el *log* del kernel, una lectura sobre el nodo de dispositivo hace que se llamen a las operaciones de apertura, lectura y cierre. El resultado de la operación de lectura tras cargar el módulo muestra que al no haber ningún mensaje previo escrito, el resultado de la lectura es un mensaje vacío. Por tanto, se procede a realizar una operación de escritura sobre el dispositivo con el comando resaltado en rojo. El mensaje a escribir corresponde

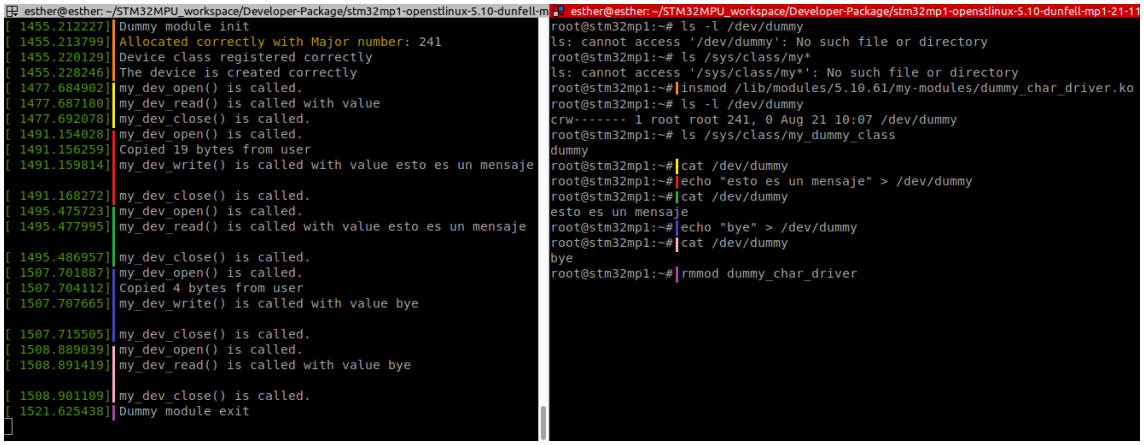


Figura 4.2: Funcionamiento del driver desarrollado

con 'esto es un mensaje', y se puede observar en el *log* del kernel como se reciben todos los caracteres. A continuación se vuelve a realizar una operación de lectura, resaltado en color verde, y se observa en la línea de comandos como se recibe el mensaje escrito en la interacción anterior. Para seguir probando el funcionamiento del driver se realiza una operación más de escritura y otra de lectura remarcadas en los colores azul y rosa.

Finalmente, se puede observar resaltado en color morado, como se descarga o elimina el módulo del kernel. Esto hace que desaparezca el nodo de dispositivo creado y la clase del dispositivo. En el *log* del kernel se muestra el mensaje procedente de la función `exit` del driver que indica que esta ha sido llamada.

4.3. Driver *GPIO* (LED)

A continuación se van a mostrar los resultados del driver implementado para encender y apagar los LEDs rojo, verde y azul, cuyo desarrollo se ha explicado en la sección 3.5.

En primer lugar se presenta la información del módulo del kernel desarrollado. En la Figura 4.3 se observan diferentes datos del módulo, como se ha visto en el módulo anterior. Como aspecto a destacar con respecto al módulo anterior, en este caso, se incluye el alias con el que se relacionan el driver y sus dispositivos compatibles, ya que se trata de un dispositivo de tipo *platform* o *plataforma*.

```
root@stm32mp1:/lib/modules/5.10.61/my-modules# modinfo ledRGB_platform_drv.ko
filename:       /lib/modules/5.10.61/my-modules/ledRGB_platform_drv.ko
description:    Platform driver that turns on/off RGB led devices.
author:         Esther Soriano
license:        GPL
alias:          of:N*T*Cessovi,RGBledsC*
alias:          of:N*T*Cessovi,RGBleds
depends:
name:           ledRGB_platform_drv
vermagic:       5.10.61 SMP preempt mod_unload modversions ARMv7 p2v8
root@stm32mp1:/lib/modules/5.10.61/my-modules#
```

Figura 4.3: Información del módulo de kernel desarrollado.

El driver desarrollado creará 3 dispositivos (uno para cada LED) en la inicialización del mismo driver, que serán accesibles bajo sus nodos en el directorio `/dev`, por lo que al cargar el driver en el kernel se deberían crear los nodos de dispositivo `/dev/ledred`, `/dev/ledgreen` y `/dev/ledblue`. Lo anterior se puede observar en la imagen derecha de la Figura 4.4, en la que se aprecia como no existe ningún nodo de dispositivo que empiece por 'led' en el directorio `/dev` cuando el driver no se encuentra cargado, pero sí existen los nodos mencionados cuando este se carga. Lo mismo se observa con el directorio que describe a cada dispositivo y se crea bajo `/sys/class/misc`.

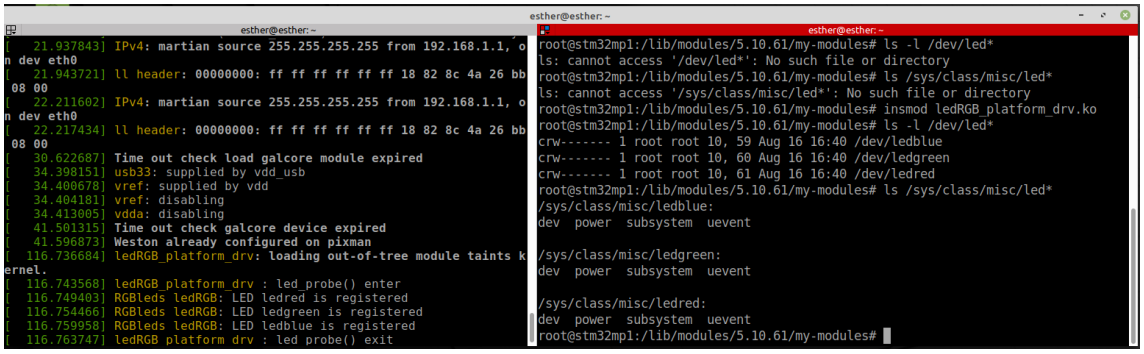


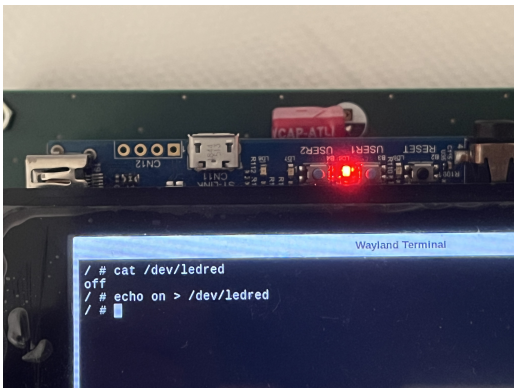
Figura 4.4: Carga del módulo en el kernel

En la imagen izquierda de la Figura 4.4, se puede observar el *log* del kernel con una marca temporal. Inicialmente en el *log* aparece todo el proceso de arranque de la placa y posteriormente, al cargar el módulo en el kernel, se observan mensajes del mismo módulo indicando que se ha llamado a la función de inicialización, así como que se han creado y registrado los dispositivos de los LEDs.

En el driver realizado se han implementado las operaciones de lectura y escritura, para poder consultar y modificar el estado de los LEDs. Para poder realizar estas operaciones es necesario, desde la capa de usuario, realizar las llamadas al sistema correspondientes, haciendo uso de los nodos de los dispositivos creados. Las operaciones de lectura y escritura sobre un dispositivo *char* se pueden realizar usando comandos de Linux como `echo` para escribir y `cat` para leer, ambas sobre los nodos de los dispositivos.

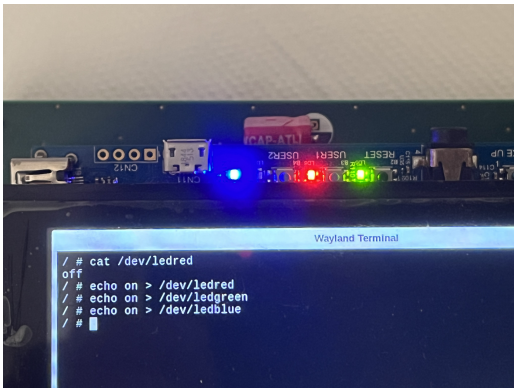
Para demostrar el funcionamiento de las operaciones de lectura y escritura implementadas, se ha conectado un teclado a la placa STM32MP157C-DK2 para escribir en la consola de Linux y visualizar en la pantalla LCD táctil adjunta los comandos que se ejecutan, observando simultáneamente los LEDs con los que se interactúa. En la Figura 4.5 se incorporan diferentes imágenes en las que se puede apreciar como se encienden y apagan los LEDs cuando se escriben los estados correspondientes sobre sus nodos de dispositivo. También se añade un ejemplo de cómo inicialmente se obtiene el estado 'off' cuando el LED rojo se encuentra apagado.

Finalmente, en la Figura 4.6 se observan los diferentes mensajes en el *log* del kernel (a la izquierda) cuando se elimina el módulo relativo al driver en la consola de Linux (a la derecha). Como se aprecia, se eliminan los dispositivos y desaparecen los nodos de los dispositivos de los respectivos directorios.



(a) Estado del led rojo y encendido del mismo

(b) Encendido del led verde



(c) Encendido del led azul

(d) Apagado del led verde

Figura 4.5: Interacciones con los dispositivos desde la capa de usuario. Encendido y apagado de los LEDs

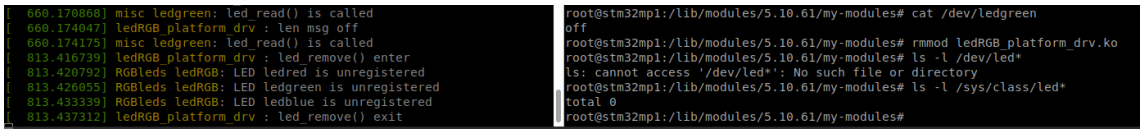


Figura 4.6: Descarga del módulo en el kernel. Eliminación de los dispositivos.

4.4. Driver UIO *GPIO* (LED)

El siguiente driver que se va a evaluar corresponde con el desarrollado en la sección 3.5.2. Este driver permite acceso a toda la memoria del banco GPIOA, pues es esta memoria la que se ha configurado en el *device tree* del dispositivo. Sin embargo, configurando la dirección correspondiente a otro banco GPIO se puede acceder a la memoria de este. Por tanto, el driver es puramente configurable. En este driver, es la aplicación de usuario la que escribe y lee sobre los registros de los GPIOs para encender y apagar los LEDs de la placa. Por tanto se va a comprobar el funcionamiento tanto del driver de kernel, como de la aplicación de usuario desarrollada.

En primer lugar, el driver UIO de kernel, al inicializarse correctamente deberá crear un dispositivo UIO, cuyo nodo se encontrará en `/dev/uioX`, siendo X un número incremental asignado por el kernel. Si no existe ningún dispositivo UIO, tomará el valor 0. Si hay alguno, tomará el valor siguiente. En la Figura 4.7 se observa en la imagen derecho como inicialmente al no estar cargado el módulo, no existe ningún dispositivo, y como al cargarlo, se crea el nodo `/dev/uio0`. En esta imagen se observan también los mensajes del *log* del kernel al cargar el driver, en los que se especifica la memoria que ha pedido el driver al sistema operativo y a la cual tiene acceso. Estos mensajes corresponden con los marcados en morado. También se observan un mensaje final al descargar el módulo, resaltado en amarillo.

En la aplicación desarrollada se puede elegir qué LED se desea controlar (antes de la compilación), siendo las opciones el LED rojo y el LED verde, ya que sus pines pertenecen al banco GPIOA, que es el que se ha configurado en esta implementación. En la Figura 4.7 se observa la ejecución de la aplicación (resaltado en un cuadro rojo), en la que únicamente se escribe el estado que se desea obtener y el LED cambia conforme a este.

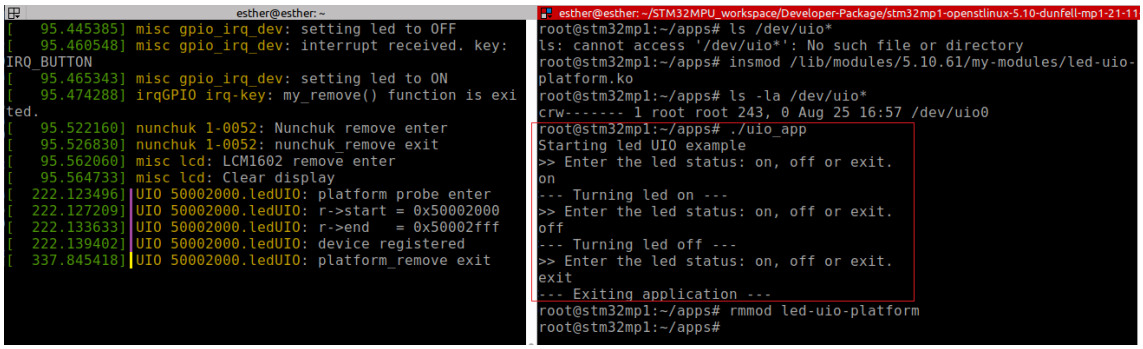


Figura 4.7: Funcionamiento del driver UIO

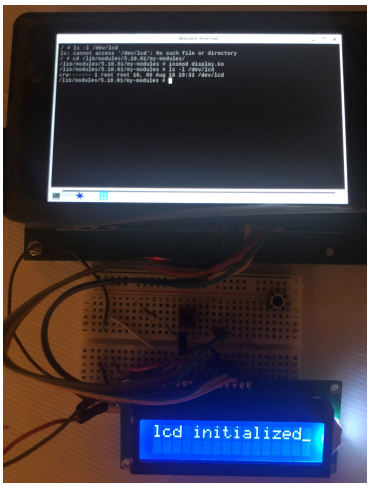
4.5. Driver LCD

En la siguiente sección se van a exponer las funcionalidades implementadas en el driver del LCD, cuyo desarrollo se ha explicado en el apartado 3.6.

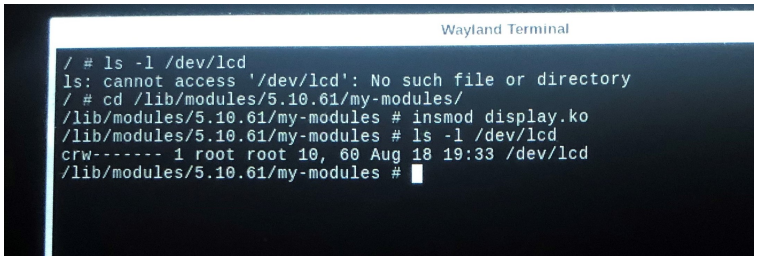
Inicialmente, tras conectar el LCD a la placa y darle alimentación, el LCD se encuentra sin inicializar, como se observa en la Figura 4.8. También se observa en la misma imagen que al no estar el módulo cargado, no existe el dispositivo lcd, ni por tanto su nodo de dispositivo. Sin embargo, tras cargar el módulo, el nodo `/dev/lcd` se encuentra disponible, como se aprecia en la Figura 4.9b.

Figura 4.8: Estado inicial del LCD

Con la carga del módulo se visualiza en la pantalla LCD el mensaje 'lcd initialized', además ver el cursor parpadeando en la posición en la que se encuentra y con las dos líneas del LCD habilitadas. Cabe recordar que esto se debe a la configuración realizada del LCD en la inicialización del driver. En las Figuras 4.9a y 4.9b se observa la carga del módulo y el mensaje correspondiente en el LCD.



(a) LCD inicializado



(b) Carga del driver

Figura 4.9: Inicialización del driver del LCD

Las funcionalidades implementadas que presenta el dispositivo LCD que se pueden usar desde la capa de usuario son las siguientes:

- Escribir mensaje en pantalla
- Borrar todo el texto de la pantalla
- Mover la posición del cursor

Para conseguir todas estas funcionalidades se ha implementado la función de escritura sobre el dispositivo y a través de una serie de códigos se identifican cada una de las tres funcionalidades. En la Tabla 4.1 se recogen estos códigos.

Funcionalidad	**Código a escribir**	**Notas**
Borrado de pantalla	'c'	Tras el borrado, el cursor se situará en la esquina superior izquierda de la pantalla (0,0).
Escritura mensaje	Mensaje	El mensaje se escribirá desde la posición del cursor en ese momento.
Cambio de la posición del cursor	'-06-01'	Los dígitos corresponden con las coordenadas X e Y de la pantalla, respectivamente.

Tabla 4.1: Código de escritura según funcionalidad en el dispositivo `/dev/lcd`

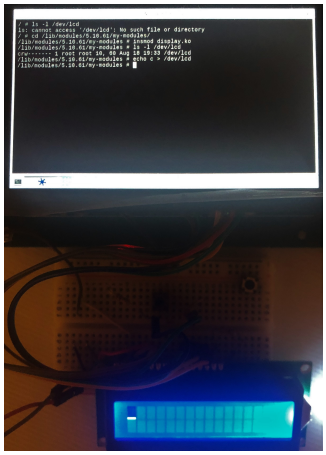
4.5.1. Control del LCD desde el espacio de usuario: comandos de Linux

A continuación se va a interactuar con el LCD desde la capa de usuario haciendo uso del comando `echo`, usado con anterioridad, y de los códigos descritos. Se puede observar en las imágenes de la Figura 4.10 como se realiza un borrado de pantalla. Además, el cursor se posiciona en la posición (0,0) que corresponde con la esquina superior izquierda.

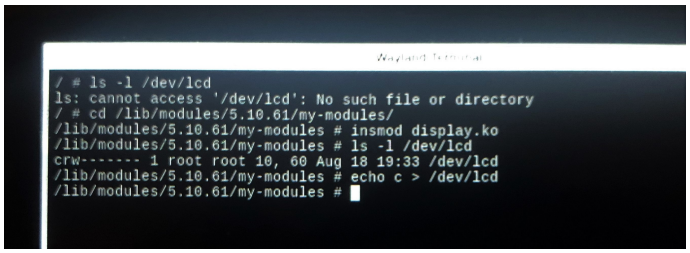
A continuación se observa en las imágenes de la Figura 4.11, como se escribe un mensaje en la pantalla del LCD. Como se puede apreciar, el mensaje se escribe desde la posición donde se encuentra el cursor.

La siguiente funcionalidad a evaluar corresponde con el cambio de la posición del cursor. Es importante destacar que a conforme se escribe en la pantalla, el cursor va incrementando su posición hacia la derecha, localizándose en la siguiente casilla al último carácter escrito. Esto forma parte de la configuración realizada en la inicialización del LCD.

Sin embargo, independientemente de que el cursor incremente su posición conforme se escriba, es posible cambiar la posición en la que se encuentra el mismo, localizándolo en



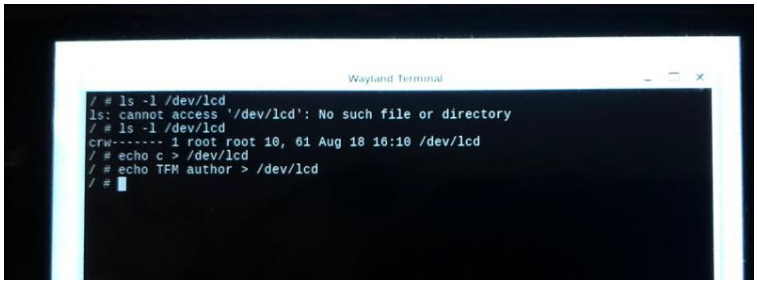
(a) LCD con pantalla borrada



(b) Vista amplificada del comando

Figura 4.10: Borrado de la pantalla LCD

(a) LCD con mensaje escrito



(b) Vista amplificada del comando

Figura 4.11: Escritura en el LCD

cualquier coordenada válida del LCD. En la Figura 4.12 se observa la cadena de caracteres escrita sobre el nodo de dispositivo para mover el cursor a las coordenadas $(0, 1)$, siendo esta posición la esquina inferior izquierda.

En esta implementación, el tamaño del LCD es de 16x2, como se ha descrito en el *device tree*, por lo que al incorporar una coordenada inválida el cursor no se moverá, y se podrá leer un mensaje de error en el *log* del kernel. En la Figura 4.13 se observa en la izquierda el *log* del kernel y en la derecha, el comando para mover el cursor a la posición $(5, 3)$, posición inválida ya que sobrepasa la altura del LCD como se indica en el mensaje de error. Además, se observa como desde la capa de usuario se recibe un mensaje *'Invalid argument'*. Esto se debe al hecho de que en el driver realizado, se devuelve el error -EINVAL, siguiendo los errores del kernel.

Finalmente, al eliminar el módulo del kernel, se observa en la siguiente Figura 4.14 que se borra la pantalla y se escribe el mensaje 'lcd drv removed'.

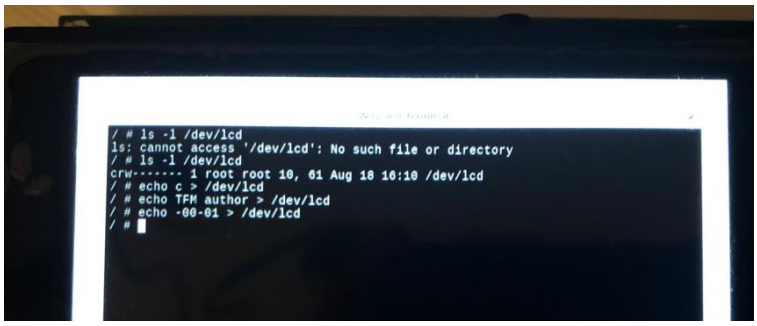
4.5.2. Control del LCD desde el espacio de usuario: aplicación en C++

Como se ha comentado en la introducción de este capítulo, para interactuar con los dispositivos es posible hacer uso de una aplicación en C, C++ u otro lenguaje que permita realizar operaciones con ficheros. En esta ocasión se va a realizar una aplicación en C++ que realizará las operaciones de apertura, escritura y cierre sobre el nodo de dispositivo del LCD.

En concreto esta aplicación realizará los siguientes pasos en bucle:

- Apertura del nodo de dispositivo, haciendo uso de la función open().
- Escritura sobre el nodo de dispositivo, haciendo uso de la función write(), para borrar la pantalla.
- Escritura sobre el nodo de dispositivo para mostrar el mensaje "Hora actual:"
- Escritura sobre el nodo de dispositivo para mover el cursor a la línea inferior.
- Obtención de la hora actual del sistema.
- Escritura sobre el nodo de dispositivo para mostrar la hora.
- Cierre del nodo de dispositivo, haciendo uso de la función close().

El código completo de la aplicación desarrollada se encuentra en el Listing A.12 del Anexo A.



(a) Cambio de la posición del cursor **(b)** Vista amplificada del comando

Figura 4.12: Movimiento del cursor del LCD

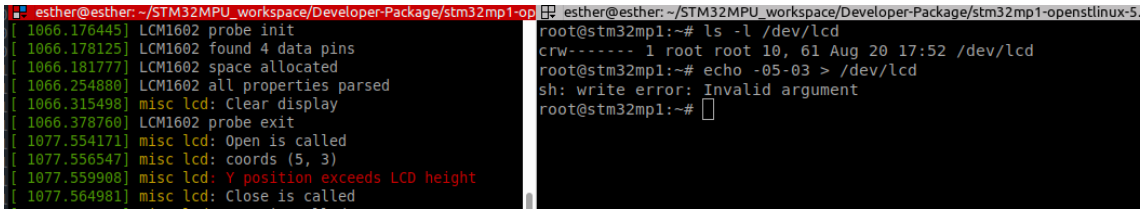
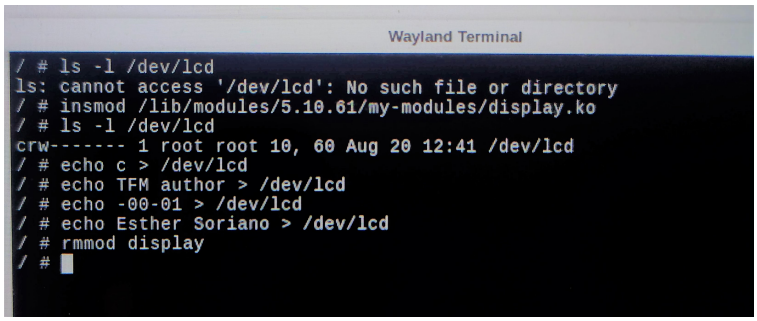


Figura 4.13: Intento de cambio de cursor a posición inválida



(a) Descarga del driver en el kernel **(b)** Vista amplificada del comando

Figura 4.14: Eliminación del módulo del kernel

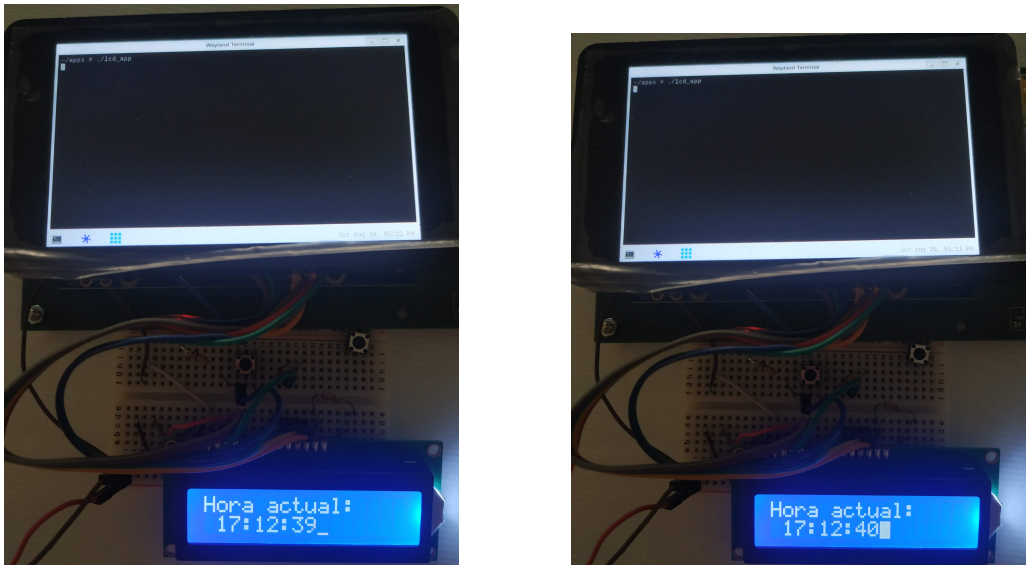


Figura 4.15: Visualización de la hora actual mediante la aplicación desarrollada

4.6. Driver GPIO con interrupciones

A continuación se van a representar los resultados del driver realizado en el apartado 3.7. El driver realizado no tiene interacción con la capa de usuario, ya que su único propósito es el de realizar la rutina programada (cambio del estado de un LED) cuando se produzca una interrupción. Cabe recordar que la configuración escogida en el *device tree* marca que una interrupción se genera cuando el estado del pin 13 , perteneciente al banco GPIOD, cambia de 0 a 1, y que el LED cuyo estado se ha de cambiar cuando esta interrupción se produzca es el del PH7, correspondiente al LED naranja.

Para comprobar el funcionamiento del driver, al tratarse de una interrupción física es complicado mostrar de una manera fiel su comportamiento. En la Figura 4.16 se puede observar la carga del driver en el kernel y los mensajes del *log* del kernel. Estos mensajes reflejan también las interrupciones que se van produciendo, que tienen lugar cuando se va pulsando el botón conectado al pin de la interrupción.

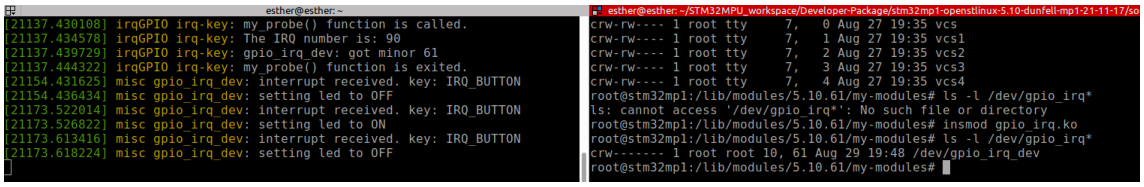


Figura 4.16: Carga y funcionamiento del driver con interrupciones

4.7. Driver cliente I2C (*Nunchuk*)

El siguiente driver a evaluar es el *Wii Nunchuk*, del tipo cliente I2C, desarrollado en la sección 3.8. El driver realizado únicamente ofrece la lectura de los datos del dispositivo *Wii Nunchuk*, por lo que la única operación a evaluar es la de lectura.

Antes de mostrar los resultados de esta operación, se muestra en la Figura 4.17 la información del módulo implementado.

```
root@stm32mp1:/lib/modules/5.10.61/my-modules# modinfo nunchuk.ko
filename:       /lib/modules/5.10.61/my-modules/nunchuk.ko
license:        GPL
author:         Esther Soriano
description:    Nunchuk I2C driver
alias:          of:N*T*CnunchukC*
alias:          of:N*T*Cnunchuk
depends:
name:           nunchuk
vermagic:       5.10.61 SMP preempt mod_unload modversions ARMv7 p2v8
root@stm32mp1:/lib/modules/5.10.61/my-modules#
```

Figura 4.17: Información del módulo de kernel desarrollado.

Al conectar físicamente los pines del *Wii Nunchuk* con los pines del expansor de Arduino de la placa STM32MP157C-DK2 y cargar el driver de kernel desarrollado para su control, el driver envía una serie de comandos específicos para 'despertar' al dispositivo, ya que se recuerda que es un esclavo I2C. En este proceso de inicialización, si físicamente el dispositivo no se encuentra conectado y por tanto no se recibe respuesta, se observa un mensaje de error en el *log* del kernel. Este mensaje se puede visualizar en la Figura 4.18, en la que cabe destacar que el errno -6, de acuerdo a los errores de Linux [10], corresponde con *ENXIO No such device or address*, lo que indica exactamente que no se encuentra el dispositivo en esa dirección. Esto hace que la inicialización del driver falle y por tanto no se cree ni el dispositivo ni su nodo.

```
[ 8000.185251] nunchuk 1-0052: Nunchuk remove enter
[ 8000.188907] nunchuk 1-0052: nunchuk_remove exit
[ 8474.423312] nunchuk 1-0052: Nunchuk probe init
[ 8474.426635] nunchuk 1-0052: error sending first handshake errno -6
[ 8474.432682] nunchuk 1-0052: Error initializing Nunchuk device.
```

Figura 4.18: Error al cargar el driver. No existe ningún dispositivo en esa dirección

Cuando el dispositivo se encuentra conectado y se carga el módulo, el driver realiza el proceso de inicialización correctamente, creando el dispositivo. En la imagen derecha de la Figura 4.19 se observa como inicialmente antes de cargar el módulo, el nodo de dispositivo `/dev/wii_nunchuk` no existe, pero al realizar la carga del driver, y tras completarse esta con éxito (tal y como se puede observar en el *log* del kernel en la imagen izquierda) el dispositivo se encuentra creado, ya que se aprecia su nodo de dispositivo.

En la misma Figura 4.19 también se puede observar como en el proceso de inicialización del driver se realiza una lectura de los datos del mismo, pudiendo visualizar en el mismo *log* del kernel los valores de los ejes del acelerómetro y del *joystick* del dispositivo *Wii*

Nunchuk. Finalmente, en la misma figura se observan los mensajes mostrados cuando se descarga el módulo del kernel.

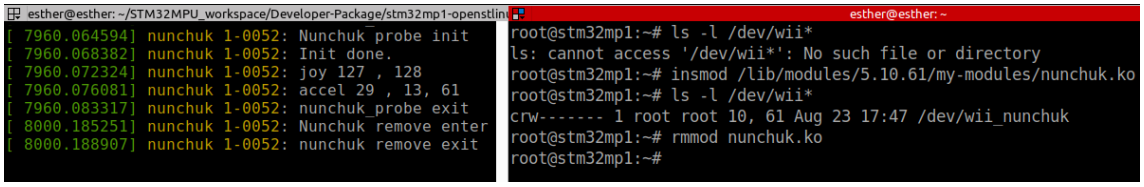


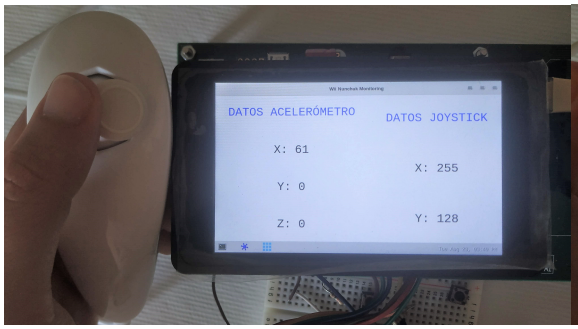
Figura 4.19: Mensajes del *log* del kernel al cargar y descargar los módulos

A continuación se van a mostrar los resultados que se obtienen cuando se realiza una operación de lectura sobre el dispositivo. Debido a la gran cantidad de datos que proporciona el dispositivo, cabe recordar que el driver implementado en la sección 3.8 los agrupa en una estructura del tipo `data`, y es esta estructura la que devuelve cuando se realiza una operación de lectura sobre el dispositivo. Por tanto, se va hacer uso de una aplicación desarrollada en C para interactuar con el dispositivo.

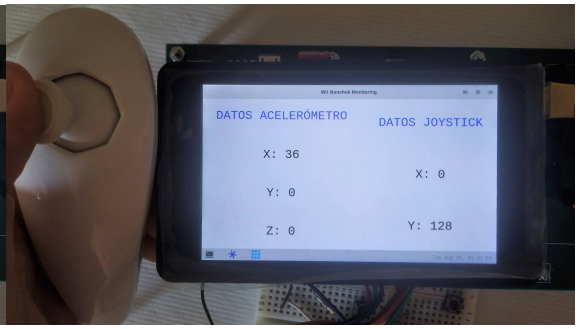
La aplicación desarrollada realiza una apertura del nodo del dispositivo y una operación de lectura sobre el mismo. Tras la lectura de los datos del *Wii Nunchuk* los imprime por pantalla. Finalmente, realiza un cierre del nodo del dispositivo. En la Figura 4.21 se observa una captura de la consola de la placa ejecutando la aplicación desarrollada. Cabe destacar que el código completo de esta aplicación se encuentra en el Listing A.18 en el Anexo A.

Para poder mostrar mejor los resultados del driver, es decir, cómo al utilizar el mando *Wii Nunchuk* los datos que se reciben del mismo son coherentes y correctos, se ha desarrollado otra aplicación en C++ que combina el uso del Nunchuk con la pantalla LCD táctil de la que dispone la placa STM32MP157C-DK2. El código de dicha aplicación se adjunta en la subsección A.6.3 del Anexo A. De esta manera, se han podido tomar fotografías de los datos del Nunchuk leídos desde la aplicación y mostrados en la pantalla LCD a la vez que se muestra la posición del joystick del dispositivo. En las imágenes de la Figura 4.20 se observan todas estas fotografías.

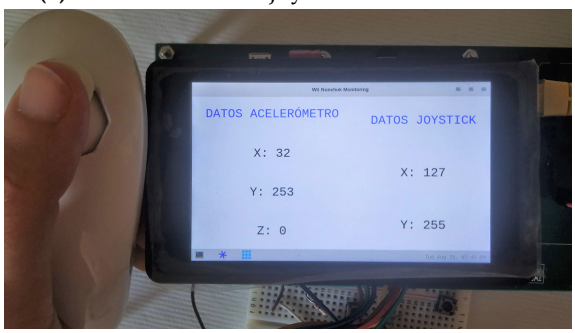
Como se puede apreciar, la posición por defecto del joystick para los ejes X e Y, respectivamente, se encuentra en torno a las coordenadas $(127, 127)$. Esto se puede observar en la Figura 4.20h. En las Figuras 4.20a y 4.20b se visualizan los valores del joystick cuando este se encuentra en el extremo derecho y en el extremo izquierdo, manteniendo la posición del eje Y como la inicial, respectivamente. En las siguientes figuras se observan el resto de los valores de los extremos del joystick.



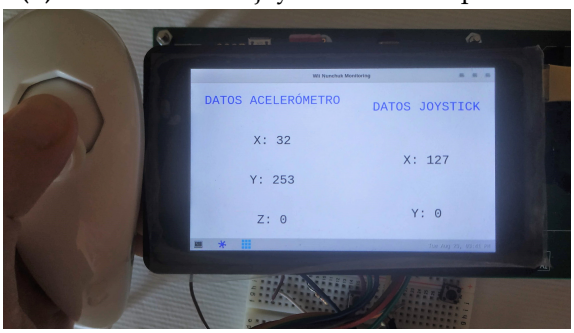
(a) Movimiento del joystick centro derecha



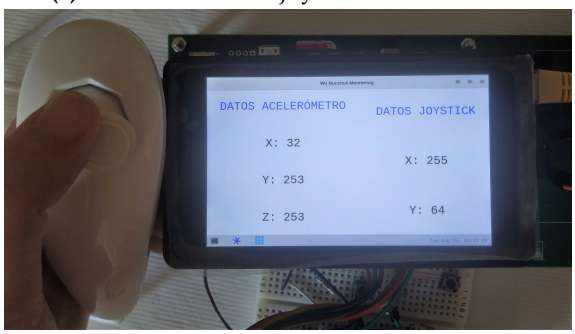
(b) Movimiento del joystick centro izquierda



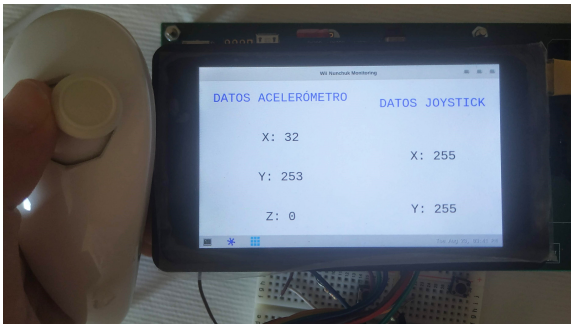
(c) Movimiento del joystick centro arriba



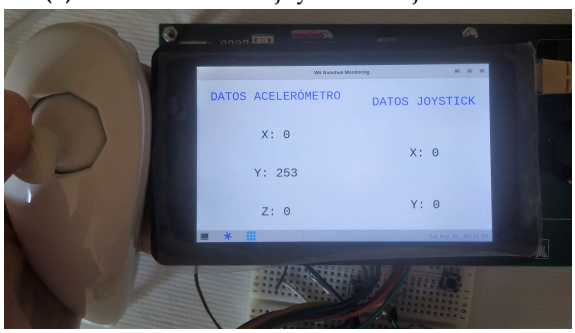
(d) Movimiento del joystick centro abajo



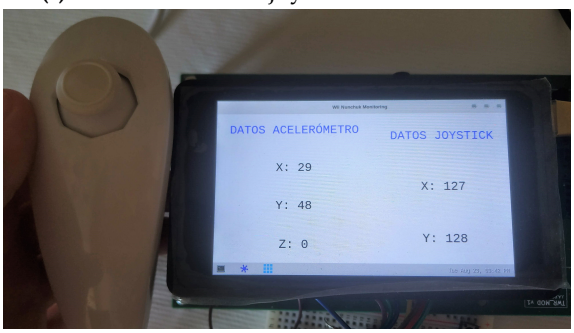
(e) Movimiento del joystick abajo derecha



(f) Movimiento del joystick arriba derecha



(g) Movimiento del joystick abajo izquierda



(h) Posición inicial del joystick

Figura 4.20: Movimientos del joystick y coordenadas leídas

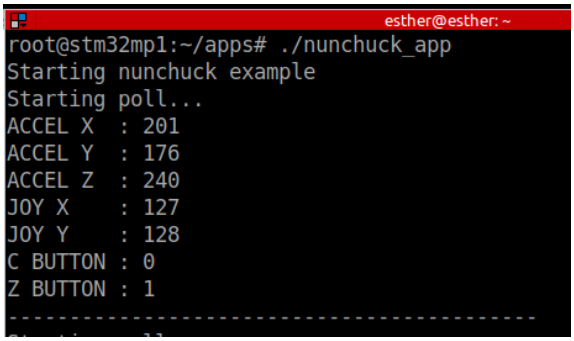


Figura 4.21: Aplicación que lee sobre el Nunchuk

Capítulo 5

Conclusiones

A modo de conclusión se va a realizar un balance de los resultados alcanzados en el proyecto.

En primer lugar, destacar el hecho de que es necesario un conocimiento previo de Linux para la realización de este proyecto, ya que los drivers realizados se han hecho a nivel de sistema operativo. También es importante mencionar que durante el desarrollo del trabajo se han adquirido una gran cantidad de conocimientos sobre Linux en relación al uso de memoria, acceso de los recursos, uso y creación de dispositivos, entre muchos otros.

Con respecto a los resultados obtenidos, se han realizado drivers de tipos muy diferentes, desde más sencillos a más complejos y con funcionalidades muy distintas. Inicialmente, se ha tratado un driver muy sencillo para explicar el concepto y la estructura de un driver, para posteriormente ir introduciendo drivers más complejos.

El driver que controla los LEDs rojo, verde y azul ha permitido introducir el concepto del *device tree*, así como la posibilidad de crear nodos dentro de un dispositivo instanciado en el mismo. También se ha podido apreciar en este driver cómo es posible realizar escrituras y lecturas directamente sobre un registro de la placa.

En el driver UIO realizado se ha podido hacer uso del tipo de driver UIO y se ha podido comprobar cómo es posible acceder a su memoria a través de una aplicación de usuario, descubriendo la gran importancia de este tipo de controladores.

Con el desarrollo del driver GPIO con interrupciones se ha realizado una introducción a la gestión de interrupciones en Linux. Se ha descubierto que es un campo muy extenso con muchísimas posibilidades pero que es posible programar interrupciones para casi cualquier driver de una manera sencilla.

Durante la realización del controlador del LCD se ha conseguido hacer uso de interfaces propias de Linux como los descriptores GPIO. También se ha aprendido a obtener toda la información necesaria del *device tree* así como se ha conseguido implementar un driver para un dispositivo externo. Esto ha supuesto obtener las hojas de datos del mismo y realizar los pasos correspondientes para poder comunicarse con la pantalla LCD. Con este driver también se ha explorado el uso de comandos a la hora de escribir sobre un dispositivo, en el que el propio driver gestiona los mensajes e identifica cada petición de

escritura sobre el dispositivo.

Por último, con el desarrollo del driver del dispositivo *Wii Nunchuk* se ha podido explorar la comunicación con un dispositivo que hace uso de un bus de comunicación, en este caso el bus I2C. Este driver ha permitido aprender a instanciar un dispositivo de este tipo en el *device tree*, así como se ha podido observar cómo se realizan las operaciones de lectura y escritura sobre el mismo dispositivo.

Es por todo lo anterior por lo que el desarrollo de los drivers realizados durante el proyecto, así como los resultados obtenidos, se valoran de una manera muy positiva, ya que no solo han servido para aprender a desarrollar este tipo de controladores sino también para conocer mejor este sistema operativo, ampliamente usado en el sector profesional.

Capítulo 6

Trabajo futuro

A continuación se propondrán actividades futuras que se podrían realizar continuando con la línea de trabajo expuesta.

En primer lugar, es posible extender el trabajo realizado explorando nuevos tipos de drivers para periféricos que usen otros buses de comunicación. Se pueden realizar drivers que hagan uso de puertos USB, SPI u otros puertos serie como UART. También existe la posibilidad de mejorar los drivers realizados, extendiendo su funcionalidad y añadiendo más configurabilidad en los drivers que así lo permitan.

Otro aspecto a investigar y desarrollar es el hecho de realizar drivers de dispositivo que dependan de otros drivers desarrollados. Esto resultaría muy interesante en el caso de tener periféricos similares pero de diferentes modelos que se comanden de distinta manera. Con lo anterior, sería posible crear drivers a nivel más abstracto que provean la funcionalidad de un tipo de driver (por ejemplo de un LCD) y diferentes drivers que implementen las interacciones con los diferentes modelos de dispositivos físicos para conseguir estas funcionalidades.

Finalmente, una propuesta futura de trabajo muy interesante es la de desarrollar los drivers a nivel de kernel para desarrollos hardware propios realizados en una FPGA. Para esto sería necesario el uso de una placa de desarrollo FPGA con un procesador con el sistema operativo Linux. Cabe destacar además que los fabricantes de FPGAs como Xilinx, cuando comercializan placas de desarrollo incluyen una versión de Linux propia para el procesador. Esta versión de Linux que ofrecen se basa en una versión del kernel de Linux concreta y a la cual suelen añadir drivers de kernel propios para extender funcionalidades o añadir configurabilidad. Estos drivers a menudo suelen interactuar con IPCores propios y que incluyen en sus diseños de referencia.

Bibliografía

[1] *GNU General Public License, version 3*. https://www.gnu.org/licenses/gpl-3.0.en.html. Last retrieved 2022-01-04. Jun. de 2007.

[2] The Linux Foundation. *LFS101x Introduction to Linux*. 7th edition.

[3] *Filesystem Hierarchy Standard, version 3*. https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html. Last retrieved 2022-01-04. Mar. de 2015.

[4] STMicroelectronics. *STM32MP1 Developer Package*. 2022. URL: https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package.

[5] Alberto Liberal de los Ríos. *Linux Driver Development for Embedded Processors*. 2018, pág. 676. ISBN: 9781729321829.

[6] STMicroelectronics. *Discovery kits with STM32MP157 MPUs - User manual*. Mar. de STMicroelectronics.

[7] *LCM1602 Datasheet*. https://datasheetspdf.com/pdf-file/746091/ETC/LCM1602K-FL-YBS/1. LCM1602 Datasheet. Abr. de 2009.

[8] STMicroelectronics. «Interrupt overview». En: *ST Wiki* (2021). URL: https://wiki.st.com/stm32mpu/wiki/Interrupt_overview.

[9] *Wii Nunchuk*. Ene. de 2005.

[10] *Errno in Linux man pages*. URL: https://man7.org/linux/man-pages/man3/errno.3.html.

Parte II

Anexos

Apéndice A

Código

A continuación se incluye el fichero Makefile para generar los binarios de los drivers de kernel a partir de sus ficheros correspondientes, los cuales se incluyen en las siguientes secciones.

Listing A.1: Makefile

```
obj-m += dummy_char_driver.o
obj-m += dummy_char_misc.o
obj-m += ledRGB_platform_drv.o
obj-m += led_uio_platform.o
obj-m += gpio_irq.o
obj-m += display.o
obj-m += nunchuk.o

KERNEL_DIR ?= $(HOME)/STM32MPU_workspace/Developer-Package/stm32mp1-openstlinux-5.10-dunfell-mp1-21-11-17/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.10.61-stm32mp-r2-r0/

all:
  make -C $(KERNEL_DIR)/build \
  ARCH=arm CROSS_COMPILE=arm-ostl-linux-gnueabi- \
  M=$(PWD) modules

clean:
  make -C $(KERNEL_DIR)/build \
  ARCH=arm CROSS_COMPILE=arm-ostl-linux-gnueabi- \
  M=$(PWD) clean
```

A.1. Dummy char Driver

A.1.1. Código del driver

Listing A.2: dummy_char_driver.c

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/device.h>
```

```
#include <linux/cdev.h>

#define DEVICE_NAME "mydev"
#define CLASS_NAME "my_dummy_class"
#define MAX_LENGTH 20U

static struct class* dummyClass;
static struct cdev my_dev;
dev_t dev;
char msg[MAX_LENGTH];

static int my_dev_open(struct inode *inode, struct file *file)
{
  pr_info("my_dev_open() is called.\n");
  return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
  pr_info("my_dev_close() is called.\n");
  return 0;
}


static ssize_t my_dev_read(struct file *file, char __user *buff, size_t count,
  loff_t *loff)
{

  ssize_t len = min(MAX_LENGTH - *loff, count);

  if(len <= 0)
    return 0;

  if(copy_to_user(buff, msg + *loff, len))
  {
    pr_info("Bad value copied");
    return -EFAULT;
  }

  pr_info("my_dev_read() is called with value %s \n", msg);

  *loff += len;
  return len;
}


static int my_dev_write(struct file *file, const char __user *buff, size_t count,
  loff_t *ppos)
{

    // Borra el contenido anterior de la variable msg.
  memset(msg, 0, MAX_LENGTH);

  if(count > MAX_LENGTH)
    count = MAX_LENGTH;

  if(copy_from_user(msg, buff, count))
  {
```

```
    pr_info("Bad value copied");
    return -EFAULT;
  }
  pr_info("Copied %d bytes from user\n", count);

  pr_info("my_dev_write() is called with value %s \n", msg);
  return count;
}



static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
  pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
  return 0;
}

// Declaración de la estructura file_operations del driver
static const struct file_operations my_dev_fops = {
  .owner    = THIS_MODULE,
  .open     = my_dev_open,
  .release    = my_dev_close,
  .read    = my_dev_read,
  .write    = my_dev_write,
  .unlocked_ioctl   = my_dev_ioctl,
};

static int __init init(void)
{
  int ret;
  dev_t dev_no;
  int Major;

  struct device* helloDevice;

  pr_info("Hello world init\n");

  // 1. Asignación dinámica de los identificadores de los dispositivos.
  // En este caso solo se requiere un único dispositivo, que tendrá el minor 0.
  ret = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);
  if (ret < 0){
    pr_info("Unable to allocate Mayor number \n");
    return ret;
  }

  // Obtener los identificadores asignados.
  Major = MAJOR(dev_no);
  dev = MKDEV(Major,0);

  pr_info("Allocated correctly with major number %d\n", Major);

  // 2. Inicialización de la estructura cdev con las operaciones que realizará el driver.
  cdev_init(&my_dev, &my_dev_fops);

  // 3. Registro del dispositivo creado en el kernel
  ret = cdev_add(&my_dev, dev, 1);
  if (ret < 0){
    unregister_chrdev_region(dev, 1);
```

```
    pr_info("Unable to add cdev\n");
    return ret;
  }

  // 4. Crear y registrar una clase para el dispositivo creado
    // (accesible en /sys/class/CLASS_NAME).
  dummyClass = class_create(THIS_MODULE, CLASS_NAME);
  if (IS_ERR(dummyClass)){
    unregister_chrdev_region(dev, 1);
    cdev_del(&my_dev);
      pr_info("Failed to register device class\n");
      return PTR_ERR(dummyClass);
  }

  pr_info("device class registered correctly\n");

  // 5. Crear un nodo de dispositivo en /dev/DEVICE_NAME
  // asociado al dispositivo creado.
  helloDevice = device_create(dummyClass, NULL, dev, NULL, DEVICE_NAME);
  if (IS_ERR(helloDevice)){
      class_destroy(dummyClass);
      cdev_del(&my_dev);
      unregister_chrdev_region(dev, 1);
      pr_info("Failed to create the device\n");
      return PTR_ERR(helloDevice);
  }
  pr_info("The device is created correctly\n");
  return 0;
}

static void __exit exit(void)
{
  // Eliminar el dispositivo.
  device_destroy(dummyClass, dev);

  // Destruir la clase del dispositivo.
  class_destroy(dummyClass);

  // Eliminar la estructura del dispositivo.
  cdev_del(&my_dev);

  // Liberar los identificadores del dispositivo.
  unregister_chrdev_region(dev, 1);
  pr_info("Dummy driver exit\n");
}

module_init(init);
module_exit(exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Esther Soriano");
MODULE_DESCRIPTION("This is a module that interacts with the system call");
```

Listing A.3: dummy_char_misc.c

```
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/module.h>

#define DEVICE_NAME "mydev"
#define MAX_LENGTH 20U

char msg[MAX_LENGTH];

static int my_dev_open(struct inode *inode, struct file *file) {
  pr_info("my_dev_open() is called.\n");
  return 0;
}

static int my_dev_close(struct inode *inode, struct file *file) {
  pr_info("my_dev_close() is called.\n");
  return 0;
}

static ssize_t my_dev_read(struct file *file, char __user *buff, size_t count,
                           loff_t *loff) {

  ssize_t len = min(MAX_LENGTH - *loff, count);

  if (len <= 0)
    return 0;

  if (copy_to_user(buff, msg + *loff, len)) {
    pr_info("Bad value copied");
    return -EFAULT;
  }

  pr_info("my_dev_read() is called with value %s \n", msg);

  *loff += len;
  return len;
}

static int my_dev_write(struct file *file, const char __user *buff,
                        size_t count, loff_t *ppos) {

  memset(msg, 0, MAX_LENGTH);

  if (count > MAX_LENGTH)
    count = MAX_LENGTH;

  if (copy_from_user(msg, buff, count)) {
    pr_info("Bad value copied");
    return -EFAULT;
  }
  pr_info("Copied %d bytes from user\n", count);

  pr_info("my_dev_write() is called with value %s \n", msg);
  return count;
}
```

```
static long my_dev_ioctl(struct file *file, unsigned int cmd,
                         unsigned long arg) {
  pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
  return 0;
}

// Declaración de la estructura file_operations del driver
static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,
    .release = my_dev_close,
    .read = my_dev_read,
    .write = my_dev_write,
    .unlocked_ioctl = my_dev_ioctl,
};

static struct miscdevice my_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &my_dev_fops,
};

static int __init init(void) {
  int ret;

  pr_info("Hello world init\n");

  // Registrar el misc device en el kernel
  ret = misc_register(&my_misc_device);
  if (ret != 0) {
    pr_err("Failed to register the misc device\n");
    return ret;
  }

  pr_info("msic device registered correctly\n");
  return 0;
}

static void __exit exit(void) {
  // Eliminar el misc device
  misc_deregister(&my_misc_device);
  pr_info("Dummy driver exit\n");
}

module_init(init);
module_exit(exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Esther Soriano");
MODULE_DESCRIPTION(
    "This is a module that interacts with the system call with misc device");
```

A.2. RGB LED Driver

A.2.1. Código del driver de kernel

Listing A.4: ledRGB_platform_drv.c

```
#define pr_fmt(fmt) KBUILD_MODNAME " : " fmt

#include <linux/clk.h> // clk_enable, clk_disable
#include <linux/fs.h> // Struct file operations.
#include <linux/io.h> // devm_ioremap, writel_relaxed
#include <linux/miscdevice.h> // misc_register
#include <linux/module.h>
#include <linux/of.h> // of_property_read_string
#include <linux/platform_device.h>
#include <linux/uaccess.h> // copy_from_user, copy_to_user

// Declare a private structure that will hold each led specific info.
struct led_device {
  struct miscdevice led_misc_device;
  u32 led_mask_set;
  u32 led_mask_clear;
  const char *led_name;
  char led_value[8];
  struct leds_device *private; // pointer to the global private struct.
};

// Declare a global private structure that will hold common information to all
// the led devices.
struct leds_device {
  u32 num_leds;
  struct clk *clk_gpioa;
  struct clk *clk_gpiod;
  struct led_device *leds[]; // pointers to each led device private struct.
};

// Allocate space for the global private struct.
static inline int sizeof_led_priv(int num_leds) {
  return sizeof(struct leds_device) + (sizeof(struct led_dev *) * num_leds);
};

/*
 * Red LD6: PA13;
 *  Green LD5: PA14;
 * Blue LD8: PD11
 * gpioa -> red and green
 *  gpiod -> blue
 */

// Physical addresses.
#define GPIOA_BASEADDR 0x50002000
#define GPIOD_BASEADDR 0x50005000

#define GPIO_MODER_OFF 0x00
#define GPIO_OTYPER_OFF 0x04
#define GPIO_PUPDR_OFF 0x0C
#define GPIO_BSRR_OFF 0x18
```

```
52
// Declare __iomem pointers that will keep virtual addresses.
static void __iomem *GPIOA_MODER_V;
static void __iomem *GPIOA_OTYPER_V;
56 static void __iomem *GPIOA_PUPDR_V;
static void __iomem *GPIOA_BSRR_V;

static void __iomem *GPIOD_MODER_V;
60 static void __iomem *GPIOD_OTYPER_V;
static void __iomem *GPIOD_PUPDR_V;
static void __iomem *GPIOD_BSRR_V;

64 // Red led -> LD6, PA13
#define GPIOA_MODER_BSRR13_SET_POS (13U)
#define GPIOA_MODER_BSRR13_CLEAR_POS (29U)
#define GPIOA_MODER_MODER13_POS (26U)
68 #define GPIOA_MODER_MODER_13_0 (0x1U << GPIOA_MODER_MODER13_POS)
#define GPIOA_MODER_MODER_13_1 (0x2U << GPIOA_MODER_MODER13_POS)
#define GPIOA_PUPDR_PUPDR_13_0 (0x1U << GPIOA_MODER_MODER13_POS)
#define GPIOA_PUPDR_PUPDR_13_1 (0x2U << GPIOA_MODER_MODER13_POS)
72
#define GPIOA_OTYPER_OTYPER_13_POS (13U)
#define GPIOA_OTYPER_OTYPER_13_Mask (0x1U << GPIOA_OTYPER_OTYPER_13_POS)

76 #define GPIOA_PA13_SET_BSSR_Mask (0x1U << GPIOA_MODER_BSRR13_SET_POS)
#define GPIOA_PA13_CLEAR_BSSR_Mask (0x1U << GPIOA_MODER_BSRR13_CLEAR_POS)

// Green led -> LD5, PA14
80 #define GPIOA_MODER_BSRR14_SET_POS (14U)
#define GPIOA_MODER_BSRR14_CLEAR_POS (30U)
#define GPIOA_MODER_MODER14_POS (28U)
#define GPIOA_MODER_MODER_14_0 (0x1U << GPIOA_MODER_MODER14_POS)
84 #define GPIOA_MODER_MODER_14_1 (0x2U << GPIOA_MODER_MODER14_POS)
#define GPIOA_PUPDR_PUPDR_14_0 (0x1U << GPIOA_MODER_MODER14_POS)
#define GPIOA_PUPDR_PUPDR_14_1 (0x2U << GPIOA_MODER_MODER14_POS)

88 #define GPIOA_OTYPER_OTYPER_14_POS (14U)
#define GPIOA_OTYPER_OTYPER_14_Mask (0x1U << GPIOA_OTYPER_OTYPER_14_POS)

#define GPIOA_PA14_SET_BSSR_Mask (0x1U << GPIOA_MODER_BSRR14_SET_POS)
92 #define GPIOA_PA14_CLEAR_BSSR_Mask (0x1U << GPIOA_MODER_BSRR14_CLEAR_POS)

// Blue led -> LD8, PD11
#define GPIOD_MODER_BSRR11_SET_POS (11U)
96 #define GPIOD_MODER_BSRR11_CLEAR_POS (27U)
#define GPIOD_MODER_MODER11_POS (22U)
#define GPIOD_MODER_MODER_11_0 (0x1U << GPIOD_MODER_MODER11_POS)
#define GPIOD_MODER_MODER_11_1 (0x2U << GPIOD_MODER_MODER11_POS)
100 #define GPIOD_PUPDR_PUPDR_11_0 (0x1U << GPIOD_MODER_MODER11_POS)
#define GPIOD_PUPDR_PUPDR_11_1 (0x2U << GPIOD_MODER_MODER11_POS)

#define GPIOD_OTYPER_OTYPER_11_POS (11U)
104 #define GPIOD_OTYPER_OTYPER_11_Mask (0x1U << GPIOD_OTYPER_OTYPER_11_POS)

#define GPIOD_PD11_SET_BSSR_Mask (0x1U << GPIOD_MODER_BSRR11_SET_POS)
#define GPIOD_PD11_CLEAR_BSSR_Mask (0x1U << GPIOD_MODER_BSRR11_CLEAR_POS)
108
static ssize_t led_write(struct file *file, const char __user *buff,
```

```
                    size_t count, loff_t *ppos) {

112 const char *led_on = "on";
    const char *led_off = "off";
    struct led_device *led_device;
    struct leds_device *leds_device;
116 struct device *dev;

    // Recover specific led_device structure (R,G,B);
    led_device =
120     container_of(file->private_data, struct led_device, led_misc_device);

    dev = led_device->led_misc_device.this_device;

124 dev_info(dev, "led_write() is called\n");

    // Recover the global private structure.
    leds_device = led_device->private;
128
    if (copy_from_user(led_device->led_value, buff, count)) {
      dev_err(dev, "Bad copied value\n");
      return -EFAULT;
132 }

    // Replace \n for \0 in led_device -> led_value
    led_device->led_value[count - 1] = '\0';
136
    dev_info(dev, "Message received from User Space : %s\n",
            led_device->led_value);

140 // Enable the GPIO ports clocks.
    clk_enable(leds_device->clk_gpioa);
    clk_enable(leds_device->clk_gpiod);

144 if (!strcmp(led_device->led_value, led_on)) {
      if (!strcmp(led_device->led_name, "ledred")) {
        writel_relaxed(led_device->led_mask_set, GPIOA_BSRR_V);
      } else if (!strcmp(led_device->led_name, "ledgreen")) {
148     writel_relaxed(led_device->led_mask_set, GPIOA_BSRR_V);
      } else if (!strcmp(led_device->led_name, "ledblue")) {
        writel_relaxed(led_device->led_mask_set, GPIOD_BSRR_V);
      } else {
152     dev_err(dev, "Bad value\n");
        return -EINVAL;
      }

156   dev_info(dev, "Led turned on");
    } else if (!strcmp(led_device->led_value, led_off)) {
      if (!strcmp(led_device->led_name, "ledred")) {
        writel_relaxed(led_device->led_mask_clear, GPIOA_BSRR_V);
160   } else if (!strcmp(led_device->led_name, "ledgreen")) {
        writel_relaxed(led_device->led_mask_clear, GPIOA_BSRR_V);
      } else if (!strcmp(led_device->led_name, "ledblue")) {
        writel_relaxed(led_device->led_mask_clear, GPIOD_BSRR_V);
164   } else {
        dev_err(dev, "Bad value\n");
        return -EINVAL;
      }
```

```
168
    dev_info(dev, "Led turned off");
  } else {
    dev_err(dev, "Bad value\n");
172   return -EINVAL;
  }

  // Disable the GPIO ports clocks.
176 clk_disable(leds_device->clk_gpioa);
  clk_disable(leds_device->clk_gpiod);

  return count;
180 }

static ssize_t led_read(struct file *file, char __user *buff, size_t count,
                        loff_t *ppos) {
184
  struct led_device *led_device;
  struct device *dev;

188 led_device =
      container_of(file->private_data, struct led_device, led_misc_device);

  dev = led_device->led_misc_device.this_device;
192
  dev_info(dev, "led_read() is called\n");

  // add \n to led_value
196 const ssize_t max_length = sizeof(led_device->led_value);
  led_device->led_value[max_length - 1] = '\n';

  ssize_t len = min(max_length - *ppos, count);
200
  if (len <= 0)
    return 0;

204 pr_info("len msg %s", led_device->led_value);

  if (copy_to_user(buff, &led_device->led_value + *ppos, len)) {
    dev_err(dev, "Failed to return led_value to user space \n");
208   return -EFAULT;
  }

  *ppos += len;
212 return len;
}

static const struct file_operations led_fops = {
216   .owner = THIS_MODULE,
    .read = led_read,
    .write = led_write,
};
220
static void configureGPIOs(void) {
  u32 GPIOA_MODER_write, GPIOD_MODER_write;
  u32 GPIOA_OTYPER_write, GPIOD_OTYPER_write;
224 u32 GPIOA_PUPDR_write, GPIOD_PUPDR_write;
```

```
    // Ensures that all leds are off when GPIOs are configured to GP output.
    // Take into account active-low leds PA13, PA14.
228 writel_relaxed(GPIOA_PA13_SET_BSSR_Mask, GPIOA_BSRR_V);
    writel_relaxed(GPIOA_PA14_SET_BSSR_Mask, GPIOA_BSRR_V);
    writel_relaxed(GPIOD_PD11_CLEAR_BSSR_Mask, GPIOD_BSRR_V);

232 // set PA13 to GP output.
    GPIOA_MODER_write = readl_relaxed(GPIOA_MODER_V);
    GPIOA_MODER_write |= GPIOA_MODER_MODER_13_0;
    GPIOA_MODER_write &= ~(GPIOA_MODER_MODER_13_1);
236 writel_relaxed(GPIOA_MODER_write, GPIOA_MODER_V);

    // set PA14 to GP output.
    GPIOA_MODER_write = readl_relaxed(GPIOA_MODER_V);
240 GPIOA_MODER_write |= GPIOA_MODER_MODER_14_0;
    GPIOA_MODER_write &= ~(GPIOA_MODER_MODER_14_1);
    writel_relaxed(GPIOA_MODER_write, GPIOA_MODER_V);

244 // set PD11 to GP output.
    GPIOD_MODER_write = readl_relaxed(GPIOD_MODER_V);
    GPIOD_MODER_write |= GPIOD_MODER_MODER_11_0;
    GPIOD_MODER_write &= ~(GPIOD_MODER_MODER_11_1);
248 writel_relaxed(GPIOD_MODER_write, GPIOD_MODER_V);

    ////////////////////

252 // set PA13 to PushPull config.
    GPIOA_OTYPER_write = readl_relaxed(GPIOA_OTYPER_V);
    GPIOA_OTYPER_write &= ~(GPIOA_OTYPER_OTYPER_13_Mask);
    writel_relaxed(GPIOA_OTYPER_write, GPIOA_OTYPER_V);
256
    // set PA14 to PushPull config.
    GPIOA_OTYPER_write = readl_relaxed(GPIOA_OTYPER_V);
    GPIOA_OTYPER_write &= ~(GPIOA_OTYPER_OTYPER_14_Mask);
260 writel_relaxed(GPIOA_OTYPER_write, GPIOA_OTYPER_V);

    // set PD11 to PushPull config.
    GPIOD_OTYPER_write = readl_relaxed(GPIOD_OTYPER_V);
264 GPIOD_OTYPER_write &= ~(GPIOD_OTYPER_OTYPER_11_Mask);
    writel_relaxed(GPIOD_OTYPER_write, GPIOD_OTYPER_V);

    ////////////////////
268
    // set PA13 PU.
    GPIOA_PUPDR_write = readl_relaxed(GPIOA_PUPDR_V);
    GPIOA_PUPDR_write |= GPIOA_PUPDR_PUPDR_13_0;
272 GPIOA_PUPDR_write &= ~(GPIOA_PUPDR_PUPDR_13_1);
    writel_relaxed(GPIOA_PUPDR_write, GPIOA_PUPDR_V);

    // set PA14 PU.
276 GPIOA_PUPDR_write = readl_relaxed(GPIOA_PUPDR_V);
    GPIOA_PUPDR_write |= GPIOA_PUPDR_PUPDR_14_0;
    GPIOA_PUPDR_write &= ~(GPIOA_PUPDR_PUPDR_14_1);
    writel_relaxed(GPIOA_PUPDR_write, GPIOA_PUPDR_V);
280
    // set PA13 PU.
    GPIOD_PUPDR_write = readl_relaxed(GPIOD_PUPDR_V);
    GPIOD_PUPDR_write |= GPIOD_PUPDR_PUPDR_11_0;
```

```
284   GPIOD_PUPDR_write &= ~(GPIOD_PUPDR_PUPDR_11_1);
      writel_relaxed(GPIOD_PUPDR_write, GPIOD_PUPDR_V);
    }

288 static int led_probe(struct platform_device *pdev) {

      int ret_val, count;
      struct device_node *child;
292
      struct leds_device *leds_device;

      struct device *dev = &pdev->dev;
296
      char led_val[8] = "off";

      pr_info("led_probe() enter\n");
300
      count = device_get_child_node_count(dev);
      if (!count)
        return -ENODEV;
304
      // Allocate space for the global private structure + pointers to led_device
      // structures.
      leds_device = devm_kzalloc(dev, sizeof_led_priv(count), GFP_KERNEL);
308   if (!leds_device)
        return -ENOMEM;

      // Get the clocks from the device tree and store them in the global structure.
312   leds_device->clk_gpioa = devm_clk_get(&pdev->dev, "GPIOA");
      if (IS_ERR(leds_device->clk_gpioa)) {
        dev_err(&pdev->dev, "failed to get clk (%ld)\n",
                PTR_ERR(leds_device->clk_gpioa));
316     return PTR_ERR(leds_device->clk_gpioa);
      }

      leds_device->clk_gpiod = devm_clk_get(&pdev->dev, "GPIOD");
320   if (IS_ERR(leds_device->clk_gpiod)) {
        dev_err(&pdev->dev, "failed to get clk (%ld)\n",
                PTR_ERR(leds_device->clk_gpiod));
        return PTR_ERR(leds_device->clk_gpiod);
324   }

      ret_val = clk_prepare(leds_device->clk_gpioa);
      if (ret_val) {
328     dev_err(&pdev->dev, "failed to prepare clk (%ld)\n", ret_val);
        return ret_val;
      }

332   ret_val = clk_prepare(leds_device->clk_gpiod);
      if (ret_val) {
        dev_err(&pdev->dev, "failed to prepare clk (%ld)\n", ret_val);
        return ret_val;
336   }

      // Enable the clocks to configure the GPIO registers.
      clk_enable(leds_device->clk_gpioa);
340   clk_enable(leds_device->clk_gpiod);
```

```
  // Get virtual addresses.
  GPIOA_MODER_V =
344     devm_ioremap(&pdev->dev, GPIOA_BASEADDR + GPIO_MODER_OFF, sizeof(u32));
  GPIOA_OTYPER_V =
      devm_ioremap(&pdev->dev, GPIOA_BASEADDR + GPIO_OTYPER_OFF, sizeof(u32));
  GPIOA_PUPDR_V =
348     devm_ioremap(&pdev->dev, GPIOA_BASEADDR + GPIO_PUPDR_OFF, sizeof(u32));
  GPIOA_BSRR_V =
      devm_ioremap(&pdev->dev, GPIOA_BASEADDR + GPIO_BSRR_OFF, sizeof(u32));

352 GPIOD_MODER_V =
      devm_ioremap(&pdev->dev, GPIOD_BASEADDR + GPIO_MODER_OFF, sizeof(u32));
  GPIOD_OTYPER_V =
      devm_ioremap(&pdev->dev, GPIOD_BASEADDR + GPIO_OTYPER_OFF, sizeof(u32));
356 GPIOD_PUPDR_V =
      devm_ioremap(&pdev->dev, GPIOD_BASEADDR + GPIO_PUPDR_OFF, sizeof(u32));
  GPIOD_BSRR_V =
      devm_ioremap(&pdev->dev, GPIOD_BASEADDR + GPIO_BSRR_OFF, sizeof(u32));
360
  configureGPIOs();

  // Disable the clocks after configuring the registers.
364 clk_disable(leds_device->clk_gpioa);
  clk_disable(leds_device->clk_gpiod);

  // Get the informatin of the device tree.
368 for_each_child_of_node(dev->of_node, child) {
    // Declare a led structure for each led device.
    struct led_device *led_device;

372   // Allocate space for each led device structure.
    led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);
    if (!led_device)
      return -ENOMEM;
376
    // Get led device name.
    of_property_read_string(child, "label", &led_device->led_name);

380 // Create a misc for each led device structure and store the specific
    // set/clear masks.
    if (strcmp(led_device->led_name, "ledred") == 0) {
      led_device->led_misc_device.minor = MISC_DYNAMIC_MINOR;
384   led_device->led_misc_device.name = led_device->led_name;
      led_device->led_misc_device.fops = &led_fops;
      led_device->led_mask_set = GPIOA_PA13_CLEAR_BSSR_Mask;
      led_device->led_mask_clear = GPIOA_PA13_SET_BSSR_Mask;
388   } else if (strcmp(led_device->led_name, "ledgreen") == 0) {
      led_device->led_misc_device.minor = MISC_DYNAMIC_MINOR;
      led_device->led_misc_device.name = led_device->led_name;
      led_device->led_misc_device.fops = &led_fops;
392   led_device->led_mask_set = GPIOA_PA14_CLEAR_BSSR_Mask;
      led_device->led_mask_clear = GPIOA_PA14_SET_BSSR_Mask;
    } else if (strcmp(led_device->led_name, "ledblue") == 0) {
      led_device->led_misc_device.minor = MISC_DYNAMIC_MINOR;
396   led_device->led_misc_device.name = led_device->led_name;
      led_device->led_misc_device.fops = &led_fops;
      led_device->led_mask_set = GPIOD_PD11_SET_BSSR_Mask;
      led_device->led_mask_clear = GPIOD_PD11_CLEAR_BSSR_Mask;
```

```
400     } else {
          dev_info(dev, "Bad device tree value\n");
          return -EINVAL;
        }
404
        // Initialize each led status to off.
        memcpy(led_device->led_value, led_val, sizeof(led_val));

408     // Register each led device.
        ret_val = misc_register(&led_device->led_misc_device);
        if (ret_val)
          return ret_val;
412
        dev_info(dev, "LED %s is registered \n", led_device->led_name);

        // each led_device will store a pointer in the global leds_device struct to
416     // recover this struct in each write() function.
        led_device->private = leds_device;

        // In the leds_device "leds" pointers points to each new led_device struct
420     // to be able to unregister misc devices in the remove() function.
        leds_device->leds[leds_device->num_leds] = led_device;
        leds_device->num_leds++;
      }
424
      // To recover leds_device global structure in remove.
      platform_set_drvdata(pdev, leds_device);

428   pr_info("led_probe() exit\n");

      return 0;
    }
432
    static int led_remove(struct platform_device *pdev) {
      int i;
      struct led_device *led_count;
436
      // recover leds_device global structure.
      struct leds_device *leds_device = platform_get_drvdata(pdev);

440   pr_info("led_remove() enter\n");

      // unregister each led device.
      for (i = 0; i < leds_device->num_leds; i++) {
444     led_count = leds_device->leds[i];
        misc_deregister(&led_count->led_misc_device);
        dev_info(&pdev->dev, "LED %s is unregistered \n", led_count->led_name);
      }
448
      // enable the clocks to configure GPIO registers.
      clk_enable(leds_device->clk_gpioa);
      clk_enable(leds_device->clk_gpiod);
452
      // Ensures that all leds are off when changes the GPIO to GP output.
      writel_relaxed(GPIOA_PA13_SET_BSSR_Mask, GPIOA_BSRR_V);
      writel_relaxed(GPIOA_PA14_SET_BSSR_Mask, GPIOA_BSRR_V);
456   writel_relaxed(GPIOD_PD11_CLEAR_BSSR_Mask, GPIOD_BSRR_V);
```

```
  // disable the clocks.
  clk_disable(leds_device->clk_gpioa);
  clk_disable(leds_device->clk_gpiod);

  pr_info("led_remove() exit\n");

  return 0;
}

static const struct of_device_id my_of_ids[] = {

    {.compatible = "essovi,RGBleds"},
    {},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver led_platform_driver = {
    .probe = led_probe,
    .remove = led_remove,
    .driver = {
        .name = "RGBleds",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }};

module_platform_driver(led_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Esther Soriano");
MODULE_DESCRIPTION("Platform driver that turns on/off RGB led devices.");
```

A.2.1.1. Dispositivo a instanciar en el DT

Listing A.5: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
    ledRGB {
    compatible = "essovi,RGBleds";
    clocks = <&rcc GPIOA>,
             <&rcc GPIOD>;
    clock-names = "GPIOA", "GPIOD";

    red {
           label = "ledred";
    };

    green {
           label = "ledgreen";
    };

    blue {
           label = "ledblue";
    };
  };
[...]
```

A.3. UIO LED Driver

A.3.1. Código del driver de kernel

Listing A.6: led_uio_platform.c

```
#include <linux/clk.h>
#include <linux/io.h> // dev_ioremap
#include <linux/module.h>
#include <linux/of.h>
#include <linux/platform_device.h> //platform_get_resource
#include <linux/uio_driver.h>

static struct uio_info the_uio_info;
static struct clk *clk;

static int my_probe(struct platform_device *pdev) {

  int ret_val;
  struct resource *r;
  struct device *dev = &pdev->dev;
  void __iomem *g_ioremap_addr;

  dev_info(dev, "platform probe enter\n");

  clk = devm_clk_get(&pdev->dev, 0);
  if (IS_ERR(clk)) {
    dev_err(dev, "failed to get clk (%ld)\n", PTR_ERR(clk));
    return PTR_ERR(clk);
  }

  ret_val = clk_prepare(clk);
  if (ret_val) {
    dev_err(dev, "failed to prepare clk (%ld)\n", ret_val);
    return ret_val;
  }

  // Get first resource from device tree.
  r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  if (!r) {
    dev_err(dev, "IORESOURCE_MEM 0 does not exist\n");
    return -EINVAL;
  }

  dev_info(dev, "r->start = 0x%08lx\n", (long unsigned int)r->start);
  dev_info(dev, "r->end = 0x%08lx\n", (long unsigned int)r->end);

  // Ioremap the memory region and get virtual address.
  g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));

  if (!g_ioremap_addr) {
    dev_err(dev, "ioremap failed\n");
    return -ENOMEM;
  }

  clk_enable(clk);
```

```
52   // Initialize uio_info struct uio_mem array.
     the_uio_info.name = "led_uio";
     the_uio_info.version = "1.0";
     the_uio_info.mem[0].memtype = UIO_MEM_PHYS;
56   the_uio_info.mem[0].addr = r->start;
     the_uio_info.mem[0].size = resource_size(r);
     the_uio_info.mem[0].name = "demo_uio_driver_hw_region";
     // Virtual driver for internal driver use.
60   the_uio_info.mem[0].internal_addr = g_ioremap_addr;

     // Register the UIO device.
     ret_val = uio_register_device(&pdev->dev, &the_uio_info);
64   if (ret_val != 0) {
       dev_info(dev, "could not register the uio device");
     }

68   dev_info(dev, "device registered");
     return 0;
}

72 static int my_remove(struct platform_device *pdev) {

     clk_disable(clk);
     uio_unregister_device(&the_uio_info);
76   dev_info(&pdev->dev, "platform_remove exit \n");

     return 0;
}
80
static const struct of_device_id my_of_ids[] = {
    {.compatible = "essovi,UIO"},
    {},
84 };

MODULE_DEVICE_TABLE(of, my_of_ids);

88 static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
92      .name = "UIO",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }};
96
module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
100 MODULE_AUTHOR("Esther Soriano");
MODULE_DESCRIPTION(
    "UIO Platform driver that turns the LED on/off without system calls.");
```

A.3.1.1. Dispositivo a instanciar en el DT

Listing A.7: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
    ledUIO {
        compatible = "essovi,UIO";
        reg = <0x50002000 0x1000>;
        clocks = <&rcc GPIOA>;
        clock-names = "GPIOA";
    }
[...]
```

A.3.2. Código del driver a nivel usuario

Listing A.8: uio-app.c

```
#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define GPIO_MODER_OFF 0x00
#define GPIO_OTYPER_OFF 0x04
#define GPIO_PUPDR_OFF 0x0c
#define GPIO_BSRR_OFF 0x18

#define RED_LED

#ifdef RED_LED
// Red led -> LD6, PIN
#define GPIOA_PIN_SET_POS (13U)
#define GPIOA_PIN_CLEAR_POS (29U)
#else
// Green led -> LD8, PA14
#define GPIOA_PIN_SET_POS (14U)
#define GPIOA_PIN_CLEAR_POS (30U)
#endif

#define GPIOA_PIN_SET_BSSR_Mask (0x1U << GPIOA_PIN_SET_POS)
#define GPIOA_PIN_CLEAR_BSSR_Mask (0x1U << GPIOA_PIN_CLEAR_POS)
#define GPIOA_PIN_MODER_MODER_0 (0x1U << (GPIOA_PIN_SET_POS)*2)
#define GPIOA_PIN_MODER_MODER_1 (0x2U << (GPIOA_PIN_SET_POS)*2)

#define UIO_SIZE_FILE "/sys/class/uio/uio0/maps/map0/size"

int main() {

  int devUIO_fd;
  unsigned int uio_size;
  void *driver_mem;
  int ret;
  char sendString[128];

  char *led_on = "on";
  char *led_off = "off";
```

```
44  char *Exit = "exit";

    printf("Starting led UIO example\n");

48  // Open file descriptor.
    devUIO_fd = open("/dev/uio0", O_RDWR | O_SYNC);
    if (devUIO_fd < 0) {
      perror("Failed opening the device \n");
52    printf("Failed opening the device \n");
      return -1;
    }

56  // Get the size to map by reading file:
    // /sys/class/uio/uio0/maps/map0/size

    // Open the file UIO_SIZE_FILE for reading.
60  FILE *size_fp = fopen(UIO_SIZE_FILE, "r");
    if (size_fp == NULL) {
      printf("Error opening UIO_SIZE_FILE \n");
      fclose(devUIO_fd);
64    return -1;
    }

    fscanf(size_fp, "0x%08X", &uio_size);
68  fclose(size_fp);

    if (uio_size < 0) {
      printf("Invalid uio_size to map");
72    close(devUIO_fd);
      return -1;
    }

76  // Do the uio_size mapping.
    driver_mem =
        mmap(NULL, uio_size, PROT_READ | PROT_WRITE, MAP_SHARED, devUIO_fd, 0);

80  if (driver_mem == MAP_FAILED) {
      printf("Error mapping devUIO");
      close(devUIO_fd);
      return -1;
84  }

    // Clear all leds.
    *(int *)(driver_mem + GPIO_BSRR_OFF) = GPIOA_PIN_SET_BSSR_Mask;
88
    uint32_t value;

    // Select led as output.
92  value = *(int *)(driver_mem + GPIO_MODER_OFF);
    value |= GPIOA_PIN_MODER_MODER_0;
    value &= ~(GPIOA_PIN_MODER_MODER_1);
    *(int *)(driver_mem + GPIO_MODER_OFF) = value;
96
    // Set to PushPull.
    value = *(int *)(driver_mem + GPIO_OTYPER_OFF);
    value &= ~(0x1U << GPIOA_PIN_SET_POS);
100 *(int *)(driver_mem + GPIO_OTYPER_OFF) = value;
```

```
  // Set PullUp resistor.
  value = *(int *)(driver_mem + GPIO_PUPDR_OFF);
  value |= GPIOA_PIN_MODER_MODER_0;
  value &= ~(GPIOA_PIN_MODER_MODER_1);
  *(int *)(driver_mem + GPIO_PUPDR_OFF) = value;

  // Control the LED.
  do {
    printf(">> Enter the led status: on, off or exit.\n");
    scanf("%[^\n]%*c", sendString);
    if (strncmp(led_on, sendString, strlen(led_on)) == 0) {
      // turn on led.
      printf("--- Turning led on --- \n");
      *(int *)(driver_mem + GPIO_BSRR_OFF) = GPIOA_PIN_CLEAR_BSSR_Mask;
    } else if (strncmp(led_off, sendString, strlen(led_off)) == 0) {
      printf("--- Turning led off --- \n");
      // turn off led.
      *(int *)(driver_mem + GPIO_BSRR_OFF) = GPIOA_PIN_SET_BSSR_Mask;
    } else if (strncmp(Exit, sendString, strlen(Exit)) == 0) {
      // EXIT APP.
      break;
    } else {
      printf("Invalid status. \n");
      return -1;
    }

  } while (1);

  // Unmap memory mapped.
  ret = munmap(driver_mem, uio_size);
  if (ret < 0) {
    printf("Error unmaping\n");
    close(devUIO_fd);
    return -1;
  }

  // Close device.
  close(devUIO_fd);
  printf("--- Exiting application ---\n");
  return 0;
}
```

A.3.2.1. Makefile de la aplicación de usuario

Listing A.9: Makefile

```
all: uio_app

app: uio_app.c
  $(CC) -o $@ $^

clean:
  rm uio_app
```

A.4. LCD Driver

A.4.1. Código del driver

Listing A.10: display.c

```
////////////////////////////////////////////////////////////////
///////// LCM1602 Character LCD driver for Linux  /////////
////////////////////////////////////////////////////////////////

#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/gpio/consumer.h>
#include <linux/miscdevice.h>
#include <linux/mod_devicetable.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/property.h>

#define COMMAND_SIZE 4

////////////////////////// Structures definitions ////////////////////////////

enum LCM1602_pin {
  /* Order does matter due to writing to GPIO array subsets! */
  PIN_D0, /* Optional */
  PIN_D1, /* Optional */
  PIN_D2, /* Optional */
  PIN_D3, /* Optional */
  PIN_D4,
  PIN_D5,
  PIN_D6,
  PIN_D7,
  PIN_RS,
  PIN_RW, /* Optional */
  PIN_EN,
  PIN_BL, /* Optional */
  PIN_NUM
};

/// Declare a private structure that will hold lcd specific info.
struct charlcd {
  const struct file_operations *ops;
  struct miscdevice charlcd_dev;
  int ifwidth;
  int height;
  int width;
  struct gpio_desc *pins[PIN_NUM];
};

////////////////////////// Function definitions ////////////////////////////////

// Driver functions definitions.
static int charlcd_open(struct inode *inode, struct file *file);
static int charlcd_release(struct inode *inode, struct file *file);
static int charlcd_write(struct file *file, const char __user *buf,
                         size_t count, loff_t *ppos);
```

```
// Internal high-level functions.
static void lcd_init(struct charlcd *lcd);
static void clear(struct charlcd *lcd);
static int move_cursor(struct charlcd *lcd, unsigned x, unsigned y);
static void print_msg(struct charlcd *lcd, char const *msg);

// Internal low-level functions.
static void write_4bitsmode(struct gpio_desc *pins[], uint8_t command,
                            unsigned rs_value);
static void print_char(struct charlcd *lcd, char c);
static void clockPulse(struct gpio_desc *pins[]);

////////////////////////////////////////////////////////////////////////////

static const struct file_operations charlcd_fops = {
    .write = charlcd_write,
    .open = charlcd_open,
    .release = charlcd_release,
};

////////////////////////// Function implementations ////////////////////////

//-------------------------- Driver functions --------------------------//

static int charlcd_open(struct inode *inode, struct file *file) {

  struct charlcd *the_charlcd;
  the_charlcd = container_of(file->private_data, struct charlcd, charlcd_dev);
  dev_info(the_charlcd->charlcd_dev.this_device, "Open is called\n");
  return 0;
}

static int charlcd_release(struct inode *inode, struct file *file) {
  struct charlcd *the_charlcd;
  the_charlcd = container_of(file->private_data, struct charlcd, charlcd_dev);
  dev_info(the_charlcd->charlcd_dev.this_device, "Close is called\n");
  return 0;
}

static int charlcd_write(struct file *file, const char __user *buf,
                         size_t count, loff_t *ppos) {

  const char __user *tmp = buf;
  char c;

  bool First = true;
  bool isCommand = false;
  unsigned CommandLen = 0;
  unsigned Command[COMMAND_SIZE];

  struct charlcd *the_charlcd;
  the_charlcd = container_of(file->private_data, struct charlcd, charlcd_dev);

  for (; count-- > 0; (*ppos)++, tmp++) {

    if (get_user(c, tmp))
      return -EFAULT;
```

```
    // If message entered is 'c';
    if (First && (c == 'c') && (count == 1)) {
      clear(the_charlcd);
      tmp++;
      First = false;
      break;
    }

    // Identifies move cursor command.
    if (c == '-' && (First || isCommand)) {
      isCommand = true;
      First = false;
      continue;
    }

    // Print character.
    if (!isCommand) {
      // Last character is \0 and it will not be printed.
      if (count == 0) {
        tmp++;
        break;
      }

      print_char(the_charlcd, c);
    }
    // Accumulates full move cursor command (only numbers). Example: '-11-01'
    else if (isCommand && (CommandLen < COMMAND_SIZE)) {
      // Save character as number.
      Command[CommandLen] = c - 48;
      CommandLen++;
    }

    First = false;
  }

  if (isCommand) {
    unsigned x, y;
    // Obtain the coordinates from digits.
    x = (Command[0] * 10) + Command[1];
    y = (Command[2] * 10) + Command[3];
    dev_info(the_charlcd->charlcd_dev.this_device, "coords (%d, %d) \n", x, y);
    if (move_cursor(the_charlcd, x, y) < 0)
      return -EINVAL;
  }

  return tmp - buf;
}

//------------------ Internal high-level functions ----------------------//

static void lcd_init(struct charlcd *lcd) {

  // Initialization sequence (for 4 bits mode)
  // - Write 0x3 3 times (waiting more that 4.1ms).
  // - Write 0x2 after that (for setting to 4 bits mode).
  // - Function set: interface, number of lines, character font.
  // - Display on.
```

```
  // - Display clear.
  // - Entry mode set.

  write_4bitsmode(lcd->pins, 0x03, 0);
  msleep(5);

  // repeat.
  write_4bitsmode(lcd->pins, 0x03, 0);
  msleep(5);

  // repeat again.
  write_4bitsmode(lcd->pins, 0x03, 0);
  ndelay(150000);

  // Set to 4 bits mode.
  write_4bitsmode(lcd->pins, 0x02, 0);

  // Command -> Function set.
  // Indicates command mode, character font size and number of lines.
  write_4bitsmode(lcd->pins, 0x02, 0); // For 4 bits mode.
  if (lcd->height > 1)
    write_4bitsmode(lcd->pins, 0x08, 0);
  else
    write_4bitsmode(lcd->pins, 0x00, 0);
  ndelay(40000);

  // Command -> Display control.
  // Sets display on, cursor on and cursor position blanking.
  write_4bitsmode(lcd->pins, 0x00, 0);
  write_4bitsmode(lcd->pins, 0x0f, 0);
  ndelay(40000);

  // Command -> Clear.
  clear(lcd);

  // Command-> Entry mode set.
  // Sets cursor move direction to right.
  write_4bitsmode(lcd->pins, 0x00, 0);
  write_4bitsmode(lcd->pins, 0x06, 0);
  msleep(1);

  // Print something.
  char letter[17] = "lcd initialized";
  print_msg(lcd, letter);
}

static void clear(struct charlcd *lcd) {
  dev_info(lcd->charlcd_dev.this_device, "Clear display\n");
  write_4bitsmode(lcd->pins, 0x00, 0);
  write_4bitsmode(lcd->pins, 0x01, 0);
  msleep(2);
}

static int move_cursor(struct charlcd *lcd, unsigned x, unsigned y) {

  if (x >= lcd->width) {
    dev_err(lcd->charlcd_dev.this_device, "X position exceeds LCD width");
    return -1;
```

```
  }

  if (y >= lcd->height) {

    dev_err(lcd->charlcd_dev.this_device, "Y position exceeds LCD height");
    return -1;
  }

  unsigned base = 0;
  if (y == 1)
    base = 0x40;

  unsigned pos;
  pos = (base | 0x80) | x;

  dev_info(lcd->charlcd_dev.this_device,
          "Cursor set to coordinates: x:%d, y:%d\n", x, y);

  // Change the cursor.
  write_4bitsmode(lcd->pins, pos >> 4, 0);
  write_4bitsmode(lcd->pins, pos & 0x0f, 0);

  msleep(1);
  return 0;
}

static void print_msg(struct charlcd *lcd, char const *msg) {
  char c;
  while ((c = *msg++)) {
    print_char(lcd, c);
  }
}

//-------------------- Internal low-level functions ----------------------//

static void clockPulse(struct gpio_desc *pins[]) {
  gpiod_set_value(pins[PIN_EN], 0);
  ndelay(1000);
  gpiod_set_value(pins[PIN_EN], 1);
  ndelay(1000);
  gpiod_set_value(pins[PIN_EN], 0);
  ndelay(100000);
}

static void write_4bitsmode(struct gpio_desc *pins[], uint8_t command,
                            unsigned rs_value) {

  gpiod_set_value(pins[PIN_RS], rs_value);

  int i;
  for (i = 0; i < 4; i++) {
    unsigned int value = (command >> i) & 0x01;
    gpiod_set_value(pins[PIN_D4 + i], value);
  }

  clockPulse(pins);
}
```

```
static void print_char(struct charlcd *lcd, char c) {
  write_4bitsmode(lcd->pins, c >> 4, 1);
  write_4bitsmode(lcd->pins, c, 1);
  ndelay(40000);
}

////////////////////////// Platform driver functions ///////////////////////

static int LCM1602_probe(struct platform_device *pdev) {
  struct device *dev = &pdev->dev;

  unsigned int i, base;
  struct charlcd *lcd;
  int ifwidth, ret;

  pr_info("LCM1602 probe init\n");

  /* Required pins */
  ifwidth = gpiod_count(dev, "data");
  if (ifwidth < 0)
    return ifwidth;

  switch (ifwidth) {
  case 4:
    base = PIN_D4;
    break;
  default:
    return -EINVAL;
  }

  pr_info("LCM1602 found %d data pins\n", ifwidth);

  lcd = devm_kzalloc(dev, sizeof(*lcd), GFP_KERNEL);
  if (!lcd)
    return -ENOMEM;

  pr_info("LCM1602 space allocated\n");

  lcd->ifwidth = ifwidth;

  // Wait for lcd to obtain enough voltage.
  msleep(50);

  // Get data-pins gpios.
  for (i = 0; i < ifwidth; i++) {
    // Gets the GPIO descriptor by searching in the device tree a GPIO that
    // matches given values. It also initializes the GPIO as output and with a
    // value of 0 -> GPIOD_OUT_LOW.
    lcd->pins[base + i] = devm_gpiod_get_index(dev, "data", i, GPIOD_OUT_LOW);

    if (IS_ERR(&lcd->pins[base + i])) {
      ret = PTR_ERR(&lcd->pins[base + i]);
      dev_err(dev, "failed to get pin (%ld)\n", ret);
      return ret;
    }
  }

  // Get enable gpio.
```

```
  // It also initializes the GPIO as output and with a value of 0 ->
  // GPIOD_OUT_LOW, that is not enabled.
  lcd->pins[PIN_EN] = devm_gpiod_get(dev, "enable", GPIOD_OUT_LOW);

  if (IS_ERR(&lcd->pins[PIN_EN])) {
    ret = PTR_ERR(&lcd->pins[PIN_EN]);
    dev_err(dev, "failed to get pin EN (%ld)\n", ret);
    return ret;
  }

  // Get rs gpio.
  // It also initializes the GPIO as output and with a value of 1 ->
  // GPIOD_OUT_HIGH, that is register select enabled.
  lcd->pins[PIN_RS] = devm_gpiod_get(dev, "rs", GPIOD_OUT_LOW);
  if (IS_ERR(&lcd->pins[PIN_RS])) {
    ret = PTR_ERR(&lcd->pins[PIN_RS]);
    dev_err(dev, "failed to get pin RS (%ld)\n", ret);
    return ret;
  }

  // Get height and width display properties.
  ret = device_property_read_u32(dev, "display-height-chars", &lcd->height);
  if (ret)
    return ret;
  ret = device_property_read_u32(dev, "display-width-chars", &lcd->width);
  if (ret)
    return ret;

  pr_info("LCM1602 all properties parsed\n");

  platform_set_drvdata(pdev, lcd);

  lcd->charlcd_dev.minor = MISC_DYNAMIC_MINOR;
  lcd->charlcd_dev.name = "lcd";
  lcd->charlcd_dev.fops = &charlcd_fops;

  ret = misc_register(&lcd->charlcd_dev);
  if (ret)
    return ret;

  // Initialize lcd.
  lcd_init(lcd);

  pr_info("LCM1602 probe exit\n");
  return 0;
}

static int LCM1602_remove(struct platform_device *pdev) {
  struct charlcd *lcd = platform_get_drvdata(pdev);

  dev_info(lcd->charlcd_dev.this_device, "LCM1602 remove enter\n");

  // Clear and display some remove message in lcd.
  clear(lcd);
  char letter[17] = "lcd drv removed";
  print_msg(lcd, letter);

  // unregister lcd device.
```

```
  misc_deregister(&lcd->charlcd_dev);

  return 0;
}

static const struct of_device_id LCM1602_of_match[] = {
   {.compatible = "lcd,LCM1602"}, {}};
MODULE_DEVICE_TABLE(of, LCM1602_of_match);

static struct platform_driver LCM1602_driver = {
   .probe = LCM1602_probe,
   .remove = LCM1602_remove,
   .driver =
      {
          .name = "LCM1602",
          .of_match_table = LCM1602_of_match,
      },
};

module_platform_driver(LCM1602_driver);
MODULE_DESCRIPTION("LCM1602 Character LCD driver");
MODULE_AUTHOR("Esther Soriano");
MODULE_LICENSE("GPL");
```

A.4.1.1. Dispositivo a instanciar en el DT

Listing A.11: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
   lcd-display {
      compatible = "lcd,LCM1602";
      data-gpios = <&gpioa 11 GPIO_ACTIVE_HIGH>,
        <&gpioa 12 GPIO_ACTIVE_HIGH>,
        <&gpioa 8 GPIO_ACTIVE_HIGH>,
        <&gpiob 10 GPIO_ACTIVE_HIGH>;

   enable-gpios = <&gpiob 12 GPIO_ACTIVE_HIGH>;
   rs-gpios = <&gpiof 7 GPIO_ACTIVE_HIGH>;

   display-width-chars = <16>;
   display-height-chars = <2>;
  };
[...]
```

A.4.2. Código de la aplicación de usuario

Listing A.12: lcd_app.cpp

```
#include <fcntl.h>
#include <iostream>
#include <ostream>
#include <stdio.h>
#include <string.h>
#include <string>
```

```
#include <time.h>
#include <unistd.h>

using namespace std;

static int lcd_clear(int fd) {

  int bytes;
  char text[2] = "c";
  bytes = write(fd, text, sizeof(text));
  if (bytes < 0) {
    perror("Failed writing to the device\n");
    return -1;
  }
  return 0;
}

static int lcd_write(int fd, char *text) {

  int bytes;
  bytes = write(fd, text, strlen(text));
  if (bytes < 0) {
    perror("Failed writing to the device\n");
    return bytes;
  }
  return 0;
}

static int lcd_move_cursor(int fd, unsigned x, unsigned y) {

  int bytes;

  // Prepare message.
  std::string msg;
  msg +=
      "-" + std::to_string(unsigned(x / 10)) + std::to_string(unsigned(x % 10));
  msg +=
      "-" + std::to_string(unsigned(y / 10)) + std::to_string(unsigned(y % 10));

  char *message = new char[msg.length() + 1];
  strcpy(message, msg.c_str());

  bytes = write(fd, message, strlen(message));

  if (bytes < 0) {
    perror("Failed writing to the device\n");
    return -1;
  }

  return 0;
}

int main() {

  while (1) {
    int fd;
    fd = open("/dev/lcd", O_WRONLY | O_SYNC);
    if (fd < 0) {
```

```
65      perror("Failed open the device\n");
        return fd;
    }

69  int err = lcd_clear(fd);
    if (err < 0)
      return err;

73  err = lcd_write(fd, "Hora actual: \n");
    if (err < 0)
      return err;

77  err = lcd_move_cursor(fd, 1, 4);
    if (err < 0)
      return err;

81  // Get system time
    time_t T = time(0);
    struct tm tm = *localtime(&T);

85  // Prepare message.
    std::string msg;
    msg += std::to_string(tm.tm_hour) + ":";
    msg += std::to_string(tm.tm_min) + ":";
89  msg += std::to_string(tm.tm_sec) + "\n";

    char *message = new char[msg.length() + 1];
    strcpy(message, msg.c_str());
93
    // Write to device node.
    err = lcd_write(fd, message);
    if (err < 0)
97    return err;

    close(fd);
    sleep(1);
101 }

  return 0;
}
```

A.4.2.1. Makefile de la aplicación de usuario

Listing A.13: Makefile

```
all: lcd_app

app: lcd_app.cpp
4   $(CXX) -o $@ $^

clean:
  rm lcd_app
```

A.5. Driver GPIO con interrupciones

A.5.1. Código del driver

Listing A.14: gpio_irq.c

```
1
#include <linux/gpio/consumer.h>
#include <linux/interrupt.h>
#include <linux/miscdevice.h>
5 #include <linux/module.h>
#include <linux/of_device.h>
#include <linux/platform_device.h>

9 static char *IRQ_NAME = "IRQ_BUTTON";

static struct irq_dev {
  struct gpio_desc *led_gpio_ntfy;
13  struct miscdevice dev;
};

/* interrupt handler */
17 static irqreturn_t hello_keys_isr(int irq, void *data) {
  struct irq_dev *intdev = data;

  dev_info(intdev->dev.this_device, "interrupt received. key: %s\n", IRQ_NAME);
21
  // Toogle led.
  int value;
  value = gpiod_get_value(intdev->led_gpio_ntfy);
25
  bool newValue;
  newValue = value & 0x01;
  gpiod_set_value(intdev->led_gpio_ntfy, !newValue);
29
  if (!newValue)
    dev_info(intdev->dev.this_device, "setting led to OFF\n");
  else
33   dev_info(intdev->dev.this_device, "setting led to ON\n");

  return IRQ_HANDLED;
}
37
static struct miscdevice gpio_irq_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "gpio_irq_dev",
41 };

static int my_probe(struct platform_device *pdev) {
  int ret_val, irq;
45  struct irq_dev *interrupt_dev;
  struct gpio_desc *gpio;
  struct gpio_desc *led_gpio;
  struct device *dev = &pdev->dev;
49
  dev_info(dev, "my_probe() function is called.\n");
```

```
  interrupt_dev = devm_kzalloc(dev, sizeof(*interrupt_dev), GFP_KERNEL);
53  if (!interrupt_dev)
      return -ENOMEM;

    /* First method to get the Linux IRQ number */
57  gpio = devm_gpiod_get(dev, 0, GPIOD_IN);
    if (IS_ERR(gpio)) {
      dev_err(dev, "gpio get failed\n");
      return PTR_ERR(gpio);
61  }

    int err;
    err = gpiod_direction_input(gpio);
65  if (err < 0) {
      dev_err(dev, "error setting input direction\n");
      return err;
    }
69
    irq = gpiod_to_irq(gpio);
    if (irq < 0)
      return irq;
73  dev_info(dev, "The IRQ number is: %d\n", irq);

    /* Get the led-gpio GPIO descriptor */
    led_gpio = devm_gpiod_get(dev, "led", GPIOD_OUT_LOW);
77  if (IS_ERR(led_gpio)) {
      dev_err(dev, "led gpio get failed\n");
      return PTR_ERR(led_gpio);
    }
81
    // Store led GPIO descriptor pointer in the internal structure.
    interrupt_dev->led_gpio_ntfy = led_gpio;

85  /* Allocate the interrupt line */
    ret_val = devm_request_irq(dev, irq, hello_keys_isr, IRQF_TRIGGER_RISING,
                               IRQ_NAME, interrupt_dev);
    if (ret_val) {
89    dev_err(dev, "Failed to request interrupt %d, error %d\n", irq, ret_val);
      return ret_val;
    }

93  interrupt_dev->dev = gpio_irq_dev;

    ret_val = misc_register(&interrupt_dev->dev);
    if (ret_val != 0) {
97    dev_err(dev, "could not register the misc device gpio_irq_dev\n");
      return ret_val;
    }

101 // To recover global structure in remove.
    platform_set_drvdata(pdev, interrupt_dev);

    dev_info(dev, "gpio_irq_dev: got minor %i\n", interrupt_dev->dev.minor);
105 dev_info(dev, "my_probe() function is exited.\n");

    return 0;
}
109
```

```
static int my_remove(struct platform_device *pdev) {
  dev_info(&pdev->dev, "my_remove() function is called.\n");

  // recover global structure.
  struct irq_dev *irq_device = platform_get_drvdata(pdev);

  misc_deregister(&irq_device->dev);
  dev_info(&pdev->dev, "my_remove() function is exited.\n");
  return 0;
}

static const struct of_device_id my_of_ids[] = {
    {.compatible = "essovi,irqGPIO"},
    {},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "irqGPIO",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }};

module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Esther Soriano");
MODULE_DESCRIPTION("This is a button interrupt platform driver");
```

A.5.1.1. Dispositivo a instanciar en el DT

Listing A.15: arch/arm/boot/dts/stm32mp157c-dk2.dts

```
[...]
irq-key {
    compatible = "essovi,irqGPIO";
    label = "PB_USER";
    gpios = <&gpiod 13 GPIO_ACTIVE_HIGH>;
    led-gpios = <&gpioa 13 GPIO_ACTIVE_HIGH>;
    interrupt-parent = <&gpiod>;
    interrupts = <13 IRQ_TYPE_EDGE_RISING>;
  };
[...]
```

A.6. Nunchuk I2C driver

A.6.1. Código del driver

Listing A.16: nunchuk.c

```
////////////////////////////////////////////////////////////////
//////// Wii Nunchuk driver for Linux ////////////
////////////////////////////////////////////////////////////////
#define pr_fmt(fmt) KBUILD_MODNAME " : " fmt

#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/i2c.h>
#include <linux/miscdevice.h>
#include <linux/module.h>

////////////////////////// Structures definitions ////////////////////////////

/// Declarar una estructura privada para guardar los datos del nunchuk.
static struct data {
  uint8_t joy_x;
  uint8_t joy_y;
  uint8_t accel_x;
  uint8_t accel_y;
  uint8_t accel_z;
  bool C_pressed;
  bool Z_pressed;
};

/// Declarar una estructura privada para contener información específica del nunchuk.
struct nunchuk {
  struct miscdevice nunchuk_dev;
  struct i2c_client *client;
};

////////////////////////// Function definitions ////////////////////////////////

// Driver functions definitions.
static int nunchuk_open(struct inode *inode, struct file *file);
static int nunchuk_release(struct inode *inode, struct file *file);
static ssize_t nunchuk_read(struct file *file, char __user *buf, size_t count,
                            loff_t *ppos);

// Internal high-level functions.
static int nunchuk_get_data(struct i2c_client *client, struct data *nunch_data);
static int nunchuk_init(struct i2c_client *client);

// Internal low-level functions.
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf,
                                  int buf_size);

////////////////////////////////////////////////////////////////////////////////

/// Define las operaciones que realizará el driver.
static const struct file_operations nunchuk_fops = {
    .open = nunchuk_open,
```

```
    .release = nunchuk_release,
    .read = nunchuk_read,
};

////////////////////////// Function implementations ////////////////////////

//--------------------------- Driver functions ---------------------------//

static int nunchuk_open(struct inode *inode, struct file *file) {
  pr_info("Open is called\n");
  return 0;
}

static int nunchuk_release(struct inode *inode, struct file *file) {
  pr_info("Close is called\n");
  return 0;
}

static ssize_t nunchuk_read(struct file *file, char __user *buf, size_t count,
                            loff_t *ppos) {

  const char __user *tmp = buf;
  struct nunchuk *nun = file->private_data;
  int ret = 0;

  struct data nunchuk_data;

  ret = nunchuk_get_data(nun->client, &nunchuk_data);
  if (ret < 0)
    return ret;

  ret = copy_to_user(buf, &nunchuk_data, sizeof(nunchuk_data));
  return ret;
}

//------------------- Internal high-level functions ----------------------//

static int nunchuk_get_data(struct i2c_client *client,
                            struct data *nunch_data) {

  u8 buf[6];
  int ret = 0;

  // Leer los registros del nunchuck
  ret = nunchuk_read_registers(client, buf, ARRAY_SIZE(buf));
  if (ret < 0) {
    dev_info(&client->dev, "Error reading the nunchuk registers.\n");
    return ret;
  }

  nunch_data->joy_x = buf[0];
  nunch_data->joy_y = buf[1];

  dev_info(&client->dev, "joy %i , %i \n", nunch_data->joy_x,
           nunch_data->joy_y);

  // Bit 0 indica si el botón Z está presionado
  nunch_data->Z_pressed = (buf[5] & BIT(0) ? false : true);
```

```
110
  // Bit 1 indica si el botón C está presionado
  nunch_data->C_pressed = (buf[5] & BIT(1)) ? false : true;

114  nunch_data->accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
  nunch_data->accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
  nunch_data->accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

118  dev_info(&client->dev, "accel %i , %i, %i \n", nunch_data->accel_x,
          nunch_data->accel_y, nunch_data->accel_z);

  return 0;
122 }

static int nunchuk_init(struct i2c_client *client) {

126  int ret;
  // Nunchuk handshakes
  u8 buf[2] = {0xf0, 0x55};
  u8 buf2[1] = {0xfb};
130
  ret = i2c_master_send(client, buf, 2);
  if (ret >= 0 && ret != 2) {
    dev_err(&client->dev, "error sending through bus\n");
134   return -EIO;
  }
  if (ret < 0) {
    dev_err(&client->dev, "error sending first handshake errno %d\n", ret);
138   return ret;
  }

  udelay(1);
142
  ret = i2c_master_send(client, buf2, 1);
  if (ret >= 0 && ret != 1) {
    dev_err(&client->dev, "error receiving\n");
146   return -EIO;
  }
  if (ret < 0) {
    dev_err(&client->dev, "error sending\n");
150   return ret;
  }

  dev_info(&client->dev, "Init done.\n");
154  udelay(1);

  // Realizar una primera medida
  struct data nunchdata;
158  ret = nunchuk_get_data(client, &nunchdata);
  if (ret < 0)
    return ret;

162  return 0;
}

//------------------- Internal low-level functions ----------------------//
166
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf,
```

```
                                int buf_size) {
  int status;

  usleep_range(10, 20);

  buf[0] = 0x00;
  status = i2c_master_send(client, buf, 1);
  if (status >= 0 && status != 1)
    return -EIO;
  if (status < 0)
    return status;

  usleep_range(10, 20);

  status = i2c_master_recv(client, buf, buf_size);
  if (status >= 0 && status != buf_size)
    return -EIO;
  if (status < 0)
    return status;

  return 0;
}

/////////////////////////// Platform driver functions ////////////////////////

static int nunchuk_probe(struct i2c_client *client) {
  struct device *dev = &client->dev;
  struct nunchuk *nunch;
  struct miscdevice nunchuk_dev;
  int ret;

  dev_info(&client->dev, "Nunchuk probe init\n");

  nunch = devm_kzalloc(dev, sizeof(*nunch), GFP_KERNEL);
  if (!nunch)
    return -ENOMEM;

  // Asociar el dispositivo cliente con la estructura interna.
  i2c_set_clientdata(client, nunch);

  // Almacenar el dispositivo cliente en la estructura interna.
  nunch->client = client;

  // Inicializar el dispostivo.
  nunchuk_dev.minor = MISC_DYNAMIC_MINOR;
  nunchuk_dev.name = "wii_nunchuk";
  nunchuk_dev.fops = &nunchuk_fops;

  // Almacenar el dispositivo en la estructura interna.
  nunch->nunchuk_dev = nunchuk_dev;

  // Inicializar el Nunchuk.
  ret = nunchuk_init(nunch->client);

  if (ret < 0) {
    dev_err(&client->dev, "Error initializing Nunchuk device.");
    return ret;
  }
```

```
226
  ret = misc_register(&nunch->nunchuk_dev);
  if (ret)
    return ret;
230
  dev_info(&client->dev, "nunchuk_probe exit\n");
  return 0;
}
234
static int nunchuk_remove(struct i2c_client *client) {

  dev_info(&client->dev, "Nunchuk remove enter\n");
238
  struct nunchuk *nunch;
  nunch = i2c_get_clientdata(client);

242  // Eliminar el dispositivo nunchuk.
  misc_deregister(&nunch->nunchuk_dev);
  dev_info(&client->dev, "nunchuk_remove exit\n");
  return 0;
246 }

// Definición de los dispositivos compatibles con el driver
static const struct of_device_id nunchuk_of_match[] = {
250    {.compatible = "nunchuk"}, {}};
MODULE_DEVICE_TABLE(of, nunchuk_of_match);

//Crear la estructura del driver i2c_driver
254 static struct i2c_driver nunchuk_driver = {
    .driver =
        {
            .name = "nunchuk",
258         .owner = THIS_MODULE,
            .of_match_table = nunchuk_of_match,
        },
    .probe_new = nunchuk_probe,
262     .remove = nunchuk_remove,
};

// Registra el driver en el bus I2C
266 module_i2c_driver(nunchuk_driver);

MODULE_DESCRIPTION("Nunchuk I2C driver");
MODULE_AUTHOR("Esther Soriano");
270 MODULE_LICENSE("GPL");
```

A.6.1.1. Dispositivo a instanciar en el DT

Listing A.17: arch/arm/boot/dts/stm32mp15xx-dkx.dtsi

```
[...]
2 &i2c5 {
  pinctrl-names = "default", "sleep";
  pinctrl-0 = <&i2c5_pins_a>;
  pinctrl-1 = <&i2c5_sleep_pins_a>;
6  i2c-scl-rising-time-ns = <185>;
```

```
  i2c-scl-falling-time-ns = <20>;
  clock-frequency = <100000>;
  /* spare dmas for other usage */
  /delete-property/dmas;
  /delete-property/dma-names;
  status = "okay";

  nunchuk: nunchuk@52 {
    compatible = "nunchuk";
    reg = <0x52>;
  };
};
[...]
```

A.6.2. Código de la aplicación de usuario

Listing A.18: nunchuk_app.c

```
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static struct data {
  uint8_t joy_x;
  uint8_t joy_y;
  uint8_t accel_x;
  uint8_t accel_y;
  uint8_t accel_z;
  bool C_pressed;
  bool Z_pressed;
};

int main() {

  struct data kernel_val;
  int fd;
  int ret;

  printf("Starting nunchuck example\n");

  // Open file descriptor.
  fd = open("/dev/wii_nunchuk", O_RDWR | O_NONBLOCK);
  if (fd < 0) {
    perror("Failed opening the device \n");
    exit(EXIT_FAILURE);
  }

  while (1) {
    puts("Starting poll...");

    ret = read(fd, &kernel_val, sizeof(kernel_val));
    if (ret < 0) {
      perror("Failed reading from the device \n");
```

```
    exit(EXIT_FAILURE);
  }

  printf("ACCEL X : %d\n", kernel_val.accel_x);
  printf("ACCEL Y : %d\n", kernel_val.accel_y);
  printf("ACCEL Z : %d\n", kernel_val.accel_z);

  printf("JOY X : %d\n", kernel_val.joy_x);
  printf("JOY Y : %d\n", kernel_val.joy_y);

  printf("C BUTTON : %d\n", kernel_val.C_pressed & 0x1);
  printf("Z BUTTON : %d\n", kernel_val.Z_pressed & 0x1);
  printf("-----------------------------------------\n");

  usleep(100);
 }

 close(fd);
 printf("Exiting application\n");
 exit(EXIT_SUCCESS);
}
```

A.6.2.1. Makefile de la aplicación de usuario

Listing A.19: Makefile

```
all: nunchuck_app

app: nunchuck_app.c
  $(CC) -o $@ $^

clean:
  rm nunchuck_app

deploy:
  scp nunchuck_app STM32MP1:/home/root/apps/.
```

A.6.3. Código de la aplicación de usuario con la pantalla táctil

Listing A.20: nunchuk_app.c

```
#include "nunchuk.h"
#include <gtk/gtk.h>
#include <math.h>
#include <string.h>
#include <string>
#include <unistd.h>
#include <vector>

static int count = 0;

static gboolean print_data_joy_x(GtkWidget *data) {

 g_return_val_if_fail(GTK_IS_LABEL(data), FALSE);
```

```
14
  Nunchuk Nunch;
  if (!Nunch.init())
    return FALSE;
18
  std::vector<uint8_t> Data;
  Data = Nunch.getJoystickData();
  if (Data.empty())
22   return FALSE;

  char *text;
  text = g_strdup_printf("X: %d", Data[0]);
26 gtk_label_set_label(GTK_LABEL(data), text);
  g_free(text);

  return TRUE;
30 }

static gboolean print_data_joy_y(GtkWidget *data) {

34 g_return_val_if_fail(GTK_IS_LABEL(data), FALSE);

  Nunchuk Nunch;
  if (!Nunch.init())
38   return FALSE;

  std::vector<uint8_t> Data;
  Data = Nunch.getJoystickData();
42 if (Data.empty())
    return FALSE;

  char *text;
46 text = g_strdup_printf("Y: %d", Data[1]);
  gtk_label_set_label(GTK_LABEL(data), text);
  g_free(text);

50 return TRUE;
}

static gboolean print_data_x(GtkWidget *data) {
54

  g_return_val_if_fail(GTK_IS_LABEL(data), FALSE);

  Nunchuk Nunch;
58 if (!Nunch.init())
    return FALSE;

  std::vector<uint8_t> Data;
62 Data = Nunch.getAccelData();
  if (Data.empty())
    return FALSE;

66 char *text;
  text = g_strdup_printf("X: %d", Data[0]);
  gtk_label_set_label(GTK_LABEL(data), text);
  g_free(text);
70

  return TRUE;
```

```
}

static gboolean print_data_y(GtkWidget *data) {

  g_return_val_if_fail(GTK_IS_LABEL(data), FALSE);

  Nunchuk Nunch;
  if (!Nunch.init())
    return FALSE;

  std::vector<uint8_t> Data;
  Data = Nunch.getAccelData();
  if (Data.empty())
    return FALSE;

  char *text;
  text = g_strdup_printf("Y: %d", Data[1]);
  gtk_label_set_label(GTK_LABEL(data), text);
  g_free(text);

  return TRUE;
}

static gboolean print_data_z(GtkWidget *data) {

  g_return_val_if_fail(GTK_IS_LABEL(data), FALSE);

  Nunchuk Nunch;
  if (!Nunch.init())
    return FALSE;

  std::vector<uint8_t> Data;
  Data = Nunch.getAccelData();
  if (Data.empty())
    return FALSE;

  char *text;
  text = g_strdup_printf("Z: %d", Data[2]);
  gtk_label_set_label(GTK_LABEL(data), text);
  g_free(text);

  return TRUE;
}

static void activate(GtkApplication *app) {
  GtkWidget *window;
  GtkWidget *box;
  GtkWidget *box_2;
  GtkWidget *box_3;

  GtkWidget *title_accel;
  GtkWidget *label_x;
  GtkWidget *label_y;
  GtkWidget *label_z;

  GtkWidget *title_joy;
  GtkWidget *label_joy_x;
  GtkWidget *label_joy_y;
```

```
130
    window = gtk_application_window_new(app);
    gtk_window_set_title(GTK_WINDOW(window), "Wii Nunchuk Monitoring");
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 200);
134
    box = gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 10);
    box_2 = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);
    box_3 = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);
138
    GdkColor color_black;
    gdk_color_parse("black", &color_black);

142 GdkColor color_blue;
    gdk_color_parse("blue", &color_blue);

    // Pango init
146 PangoFontDescription *pfd;
    pfd = pango_font_description_from_string("Times New Roman");
    pango_font_description_set_size(pfd, 25 * PANGO_SCALE);

150 title_accel = gtk_label_new("DATOS ACELERÓMETRO");
    label_x = gtk_label_new("0");
    label_y = gtk_label_new("1");
    label_z = gtk_label_new("2");
154
    title_joy = gtk_label_new("DATOS JOYSTICK");
    label_joy_x = gtk_label_new("0");
    label_joy_y = gtk_label_new("0");
158
    gtk_widget_modify_font(title_accel, pfd);
    gtk_widget_modify_fg(title_accel, GTK_STATE_NORMAL, &color_blue);

162 gtk_widget_modify_font(label_x, pfd);
    gtk_widget_modify_fg(label_x, GTK_STATE_NORMAL, &color_black);

    gtk_widget_modify_font(label_y, pfd);
166 gtk_widget_modify_fg(label_y, GTK_STATE_NORMAL, &color_black);

    gtk_widget_modify_font(label_z, pfd);
    gtk_widget_modify_fg(label_z, GTK_STATE_NORMAL, &color_black);
170
    gtk_widget_modify_font(title_joy, pfd);
    gtk_widget_modify_fg(title_joy, GTK_STATE_NORMAL, &color_blue);

174 gtk_widget_modify_font(label_joy_x, pfd);
    gtk_widget_modify_fg(label_joy_x, GTK_STATE_NORMAL, &color_black);

    gtk_widget_modify_font(label_joy_y, pfd);
178 gtk_widget_modify_fg(label_joy_y, GTK_STATE_NORMAL, &color_black);

    // Call it every 250 ms.
    g_timeout_add(250, G_SOURCE_FUNC(print_data_x), label_x);
182 g_timeout_add(250, G_SOURCE_FUNC(print_data_y), label_y);
    g_timeout_add(250, G_SOURCE_FUNC(print_data_z), label_z);
    g_timeout_add(250, G_SOURCE_FUNC(print_data_joy_x), label_joy_x);
    g_timeout_add(250, G_SOURCE_FUNC(print_data_joy_y), label_joy_y);
186
    gtk_container_add(GTK_CONTAINER(window), box);
```

```
  gtk_box_pack_start(GTK_BOX(box_2), title_accel, 1, 1, 10);
  gtk_box_pack_start(GTK_BOX(box_2), label_x, 1, 1, 10);
  gtk_box_pack_start(GTK_BOX(box_2), label_y, 1, 1, 10);
  gtk_box_pack_start(GTK_BOX(box_2), label_z, 1, 1, 10);

  gtk_box_pack_start(GTK_BOX(box_3), title_joy, 1, 1, 10);
  gtk_box_pack_start(GTK_BOX(box_3), label_joy_x, 1, 1, 10);
  gtk_box_pack_start(GTK_BOX(box_3), label_joy_y, 1, 1, 10);

  gtk_box_pack_start(GTK_BOX(box), box_2, 1, 1, 0);
  gtk_box_pack_start(GTK_BOX(box), box_3, 1, 1, 0);

  gtk_widget_show_all(window);
}

int main(int argc, char **argv) {

  GtkApplication *app;
  int status;

  app = gtk_application_new("org.gtk.example", G_APPLICATION_FLAGS_NONE);
  g_signal_connect(app, "activate", G_CALLBACK(activate), NULL);
  status = g_application_run(G_APPLICATION(app), FALSE, NULL);
  g_object_unref(app);

  return status;
}
```

Listing A.21: nunchuk_app.c

```
#ifndef INCLUDE_NUNCHUK
#define INCLUDE_NUNCHUK

#include <stdint.h>
#include <vector>

struct NunchukData {
  uint8_t xJoy;
  uint8_t yJoy;
  uint8_t xAcc;
  uint8_t yAcc;
  uint8_t zAcc;
  bool zButton;
  bool cButton;
};

class Nunchuk {
public:
  Nunchuk(){};

  ~Nunchuk();

  bool init();

  bool getData(NunchukData &Data);

  std::vector<uint8_t> getAccelData();
```

```
29   std::vector<uint8_t> getJoystickData();

     bool isCButtonPressed();

33   bool isZButtonPressed();

private:
  int fd;
37 };

#endif
```

Listing A.22: nunchuk_app.c

```
1  #include "nunchuk.h"
   #include <fcntl.h>
   #include <stdio.h>
   #include <unistd.h>
5
   Nunchuk::~Nunchuk() { close(fd); }

   bool Nunchuk::init() {
9    fd = open("/dev/wii_nunchuk", O_RDONLY | O_NONBLOCK);
     if (fd < 0) {
       perror("Failed open the device\n");
       return false;
13   }

     return true;
   }
17
   bool Nunchuk::getData(NunchukData &Data) {
     int err;

21   err = read(fd, &Data, sizeof(NunchukData));
     if (err < 0) {
       perror("Failed reading from the device\n");
       return false;
25   }

     return true;
   }
29
   std::vector<uint8_t> Nunchuk::getAccelData() {
     NunchukData Data;

33   if (!getData(Data))
       return {};

     return std::vector<uint8_t>({Data.xAcc, Data.yAcc, Data.zAcc});
37 }

   std::vector<uint8_t> Nunchuk::getJoystickData() {
     NunchukData Data;
41
     if (!getData(Data))
       return {};
```

```
  return std::vector<uint8_t>({Data.xJoy, Data.yJoy});
}

bool Nunchuk::isCButtonPressed() {
  NunchukData Data;

  if (!getData(Data))
    return false;

  return Data.cButton;
}

bool Nunchuk::isZButtonPressed() {
  NunchukData Data;

  if (!getData(Data))
    return false;

  return Data.zButton;
}
```

A.6.3.1. Makefile de la aplicación de usuario

Listing A.23: Makefile

```
PROG = nunchuk_gtk_app
SRCS += nunchuk_gtk_app.cpp
SRCS += nunchuk.cpp

CLEANFILES = $(PROG)

CXXFLAGS += -Wall $(shell pkg-config --cflags gtk+-3.0)
LDFLAGS += $(shell pkg-config --libs gtk+-3.0)

all : $(PROG)

$(PROG) : $(SRCS)
  $(CXX) -o $@ $^ $(CXXFLAGS) $(LDFLAGS)

clean:
  rm -rf $(CLEANFILES) $(patsubst %.cpp, %.o, $(SRCS))

deploy:
  scp $(PROG) STM32MP1:/home/root/apps/.
```

Apéndice B

Registros de STM32MP157C-DK2

13.4 GPIO registers

This section gives a detailed description of the GPIO registers.

For a summary of register bits, register address offsets and reset values, refer to *Table 84*.

The peripheral registers can be written in word, half word or byte mode.

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

13.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A to K, Z)

Address offset: 0x04

Reset value: 0x0000 0000

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
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OT[15:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain

13.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A to K, Z)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [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
OSPEEDR7 [1:0]		OSPEEDR6 [1:0]		OSPEEDR5 [1:0]		OSPEEDR4 [1:0]		OSPEEDR3 [1:0]		OSPEEDR2 [1:0]		OSPEEDR1 [1:0]		OSPEEDR0 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **OSPEEDR[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 output speed.

00: Low speed
01: Medium speed
10: High speed
11: Very high speed

Note: Refer to the product datasheets for the values of OSPEEDRy bits versus V_{DD} range and external load.

13.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A to K, Z)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **PUPDR[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 pull-up or pull-down

00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

13.4.5 GPIO port input data register (GPIOx_IDR) (x = A to K, Z)

Address offset: 0x10

Reset value: 0x0000 XXXX

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDR[15:0]:** Port x input data I/O pin y (y = 15 to 0)
These bits are read-only. They contain the input value of the corresponding I/O port.

13.4.6 GPIO port output data register (GPIOx_ODR) (x = A to K, Z)

Address offset: 0x14

Reset value: 0x0000 0000

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODR[15:0]:** Port output data I/O pin y (y = 15 to 0)
These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR register (x = A..F).

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

13.4.8 GPIO port configuration lock register (GPIOx_LCKR) (x = A to K, Z)

This register is used to lock the configuration of the port bits when a correct write sequence is applied to bit 16 (LCKK). The value of bits [15:0] is used to lock the configuration of the GPIO. During the write sequence, the value of LCKR[15:0] must not change. When the LOCK sequence has been applied on a port bit, the value of this port bit can no longer be modified until the next MCU reset or peripheral reset.

Note: A specific write sequence is used to write to the GPIOx_LCKR register. Only word access (32-bit long) is allowed during this locking sequence.

Each lock bit freezes a specific configuration register (control and alternate function registers).

Address offset: 0x1C

Reset value: 0x0000 0000

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.	LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Apéndice C

Registros del Wii Nunchuk

Data byte receive								Address
Joystick X								0x00
Joystick Y								0x01
Accelerometer X (bit 9 to bit 2 for 10-bit resolution)								0x02
Accelerometer Y (bit 9 to bit 2 for 10-bit resolution)								0x03
Accelerometer Z (bit 9 to bit 2 for 10-bit resolution)								0x04
Accel. Z bit 1	Accel. Z bit 0	Accel. Y bit 1	Accel. Y bit 0	Accel. X bit 1	Accel. X bit 0	C-button	Z-button	0x05

Byte 0x00 : X-axis data of the joystick

Byte 0x01 : Y-axis data of the joystick

Byte 0x02 : X-axis data of the accellerometer sensor

Byte 0x03 : Y-axis data of the accellerometer sensor

Byte 0x04 : Z-axis data of the accellerometer sensor

Byte 0x05 : bit 0 as Z button status - 0 = pressed and 1 = release

bit 1 as C button status - 0 = pressed and 1 = release

bit 2 and 3 as 2 lower bit of X-axis data of the accellerometer sensor

bit 4 and 5 as 2 lower bit of Y-axis data of the accellerometer sensor

bit 6 and 7 as 2 lower bit of Z-axis data of the accellerometer sensor

Figura C.1: Registros del mando Wii Nunchuk