Linux Kernel Module Programming II

...

Sistemas de Computación (IComp + IE)

¿ Quién es Werner Almesberger ?

Quizás lo podamos descubrir vinculado a un módulo de Linux asociado a los sistemas de archivos (`fs` / `fat`) con los que el SO puede trabajar..!!

Podemos empezar la búsqueda en `/lib/modules/`

¿ Qué es un driver ?

Se trata de un software que permite al sistema operativo interactuar con un periférico, creando una abstracción del hardware y proporcionando una interfaz -posiblemente estandarizada- para utilizarlo. Se puede graficar como un manual de instrucciones que indica cómo controlar y comunicarse con un dispositivo en particular.

Se trata de una pieza esencial, sin la cual no se podría usar el hardware.

En un ambiente de IoT, CPS, Industrias 4.0, etc...

Driver... ¿Quién los construye ?

Habitualmente son los fabricantes del hardware quienes escriben sus drivers, ya que conocen mejor el funcionamiento interno de cada hardware, pero también se encuentran controladores libres, por ejemplo en los sistemas operativos libres.

En este caso, los creadores no son de la empresa fabricante, aunque a veces hay una cooperación con ellos, cosa que facilita el desarrollo. Si no la hay, el procedimiento necesita de ingeniería inversa y otros métodos con riesgos legales.

¿ Cuantos tipos podemos encontrar ?

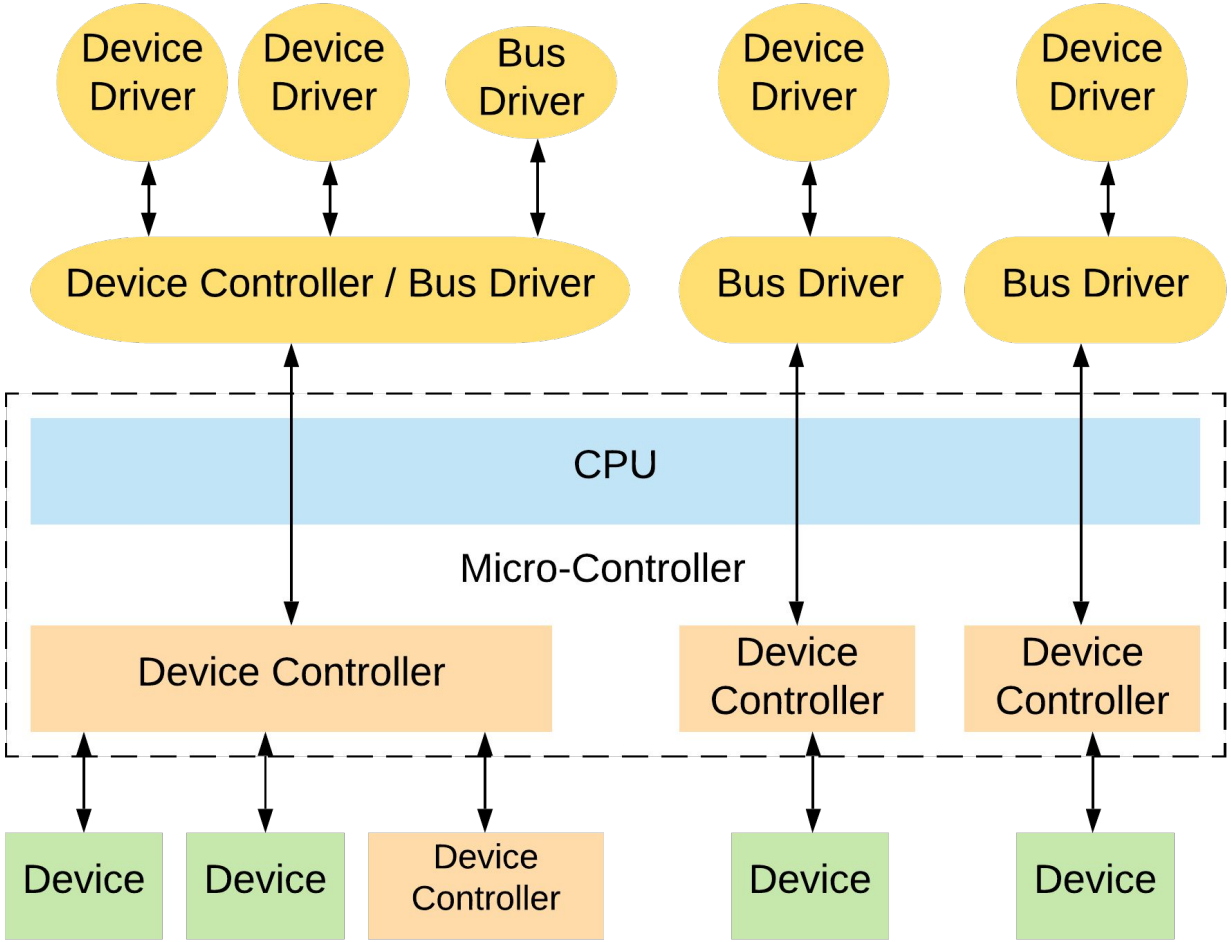
Si bien un conductor es una persona, también podría ser un sistema automático, quizás monitoreado por una persona.

Del mismo modo, un “device driver” podría ser una pieza de software u otro periférico/device, posiblemente controlado por un software. En estos casos se lo denomina “device controller”.

Un “device controller” es un device en sí mismo y, por lo tanto, muchas veces también necesita un driver, comúnmente conocido como “bus controller”.

Con frecuencia encontramos embebidos que son microcontroladores, que no son más que CPU + varios “device controller” integrados en un solo chip.

Sobre drivers y buses



Sobre drivers y buses

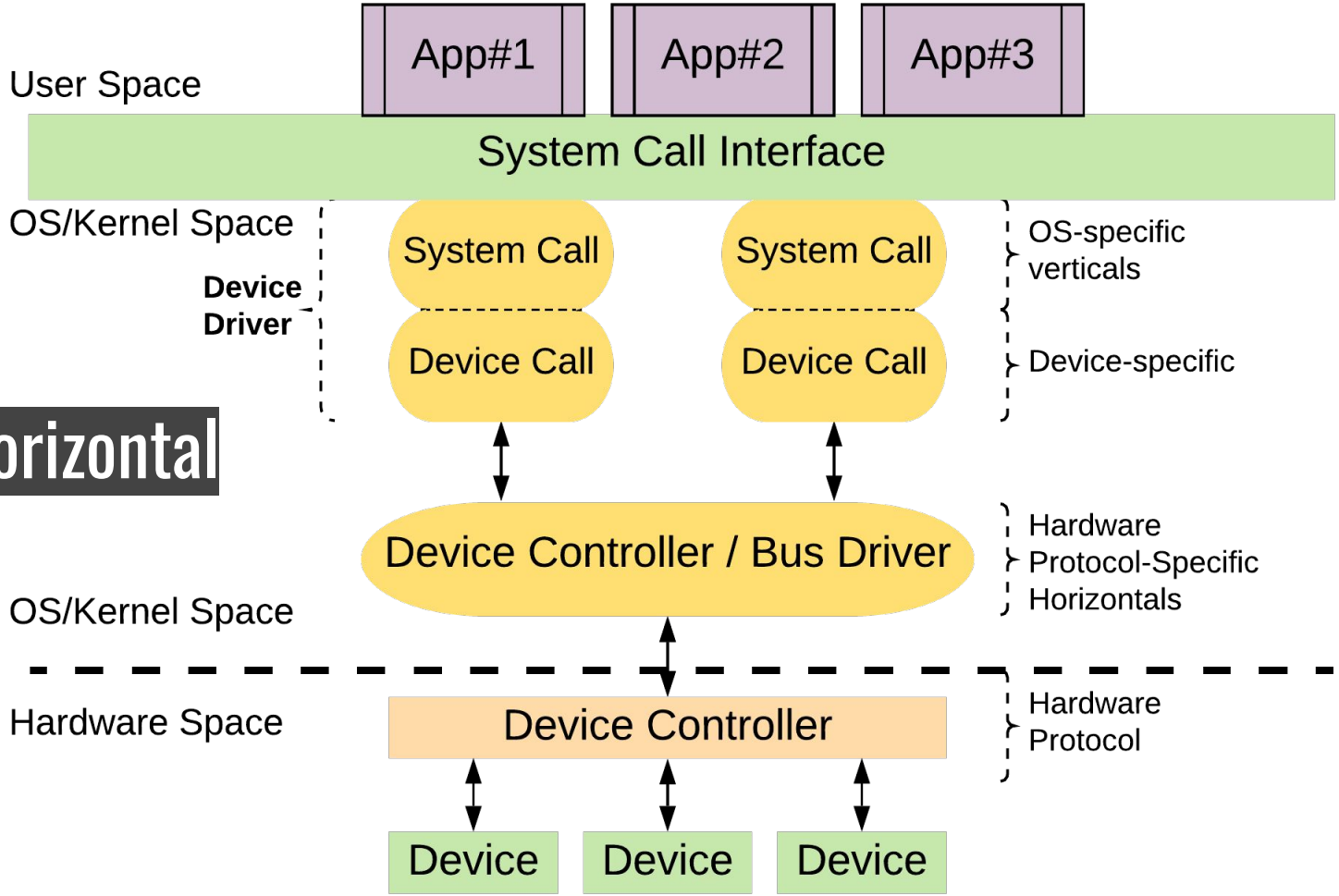
Los “bus driver” proporcionan una interfaz específica de hardware para los correspondientes protocolos de hardware y son las capas de software (horizontales) más bajas de un sistema operativo.

Sobre ellos se encuentran los “device driver”, que operan sobre los dispositivos subyacentes utilizando las interfaces de capa horizontal y, por lo tanto, son específicos del dispositivo.

La idea de escribir estos “drivers” es proporcionar una abstracción del hardware para las aplicaciones de usuario a través del sistema operativo. Ofrecer una interfaz común para el usuario que será específica para cada sistema operativo.

En resumen, un “device driver” tiene dos partes: i) Una específica del dispositivo y ii) Otra específica del sistema operativo.

Una mirada horizontal

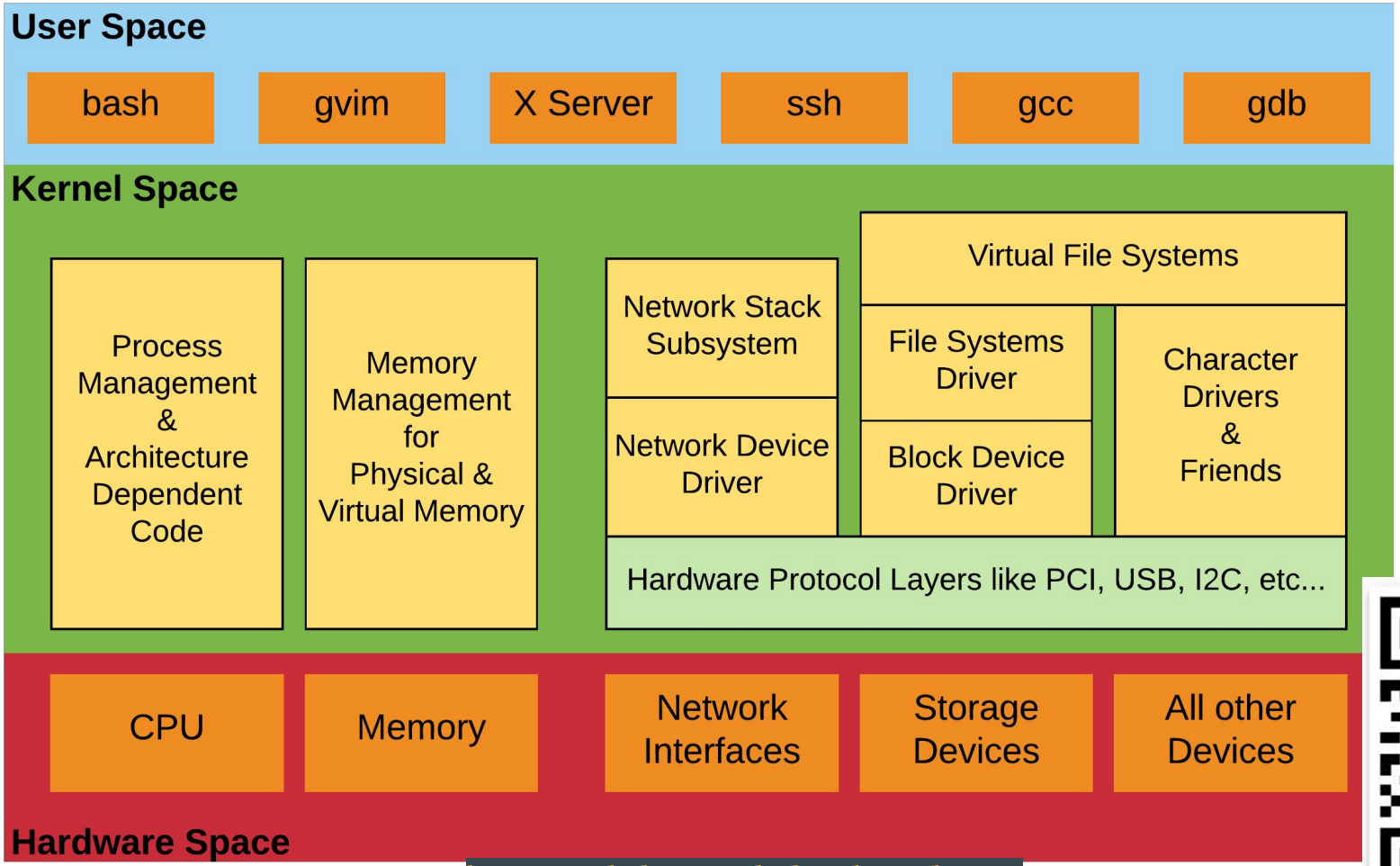


Una mirada vertical

Basado en la interfaz específica del sistema operativo de un “driver”, en Linux un “driver” se clasifica en general en tres verticales:

- Orientado a paquetes o vertical “Network”
- Orientado a bloques o vertical “Storage”
- Orientado a bytes o vertical “Character”

El directorio `/dev` contiene los archivos de dispositivos especiales para todos los dispositivos hardware.



Un punto de vista vertical

https://makelinux.github.io/kernel/map/

Algunas primeras conclusiones

Un “device driver” es un componente de software que controla un dispositivo y se pueden clasificar en driver de paquetes, bloques y character.

En caso de que solo maneje otra pieza de software, lo llamamos solo “driver”. Algunos ejemplos son los controladores del sistema de archivos, usbcore, etc.

Todos los “device driver” son “drivers” pero no todos los “drivers” son “device driver”.

En el vertical “character” se ubica el grupo mayoritario de drivers.!!

¿ Por qué tanto interés con los CDD ?

Por ejemplo, controladores de puertos seriales, controladores de audio, controladores de video, controladores de cámara y controladores básicos de E/S, son todos “character device driver”.

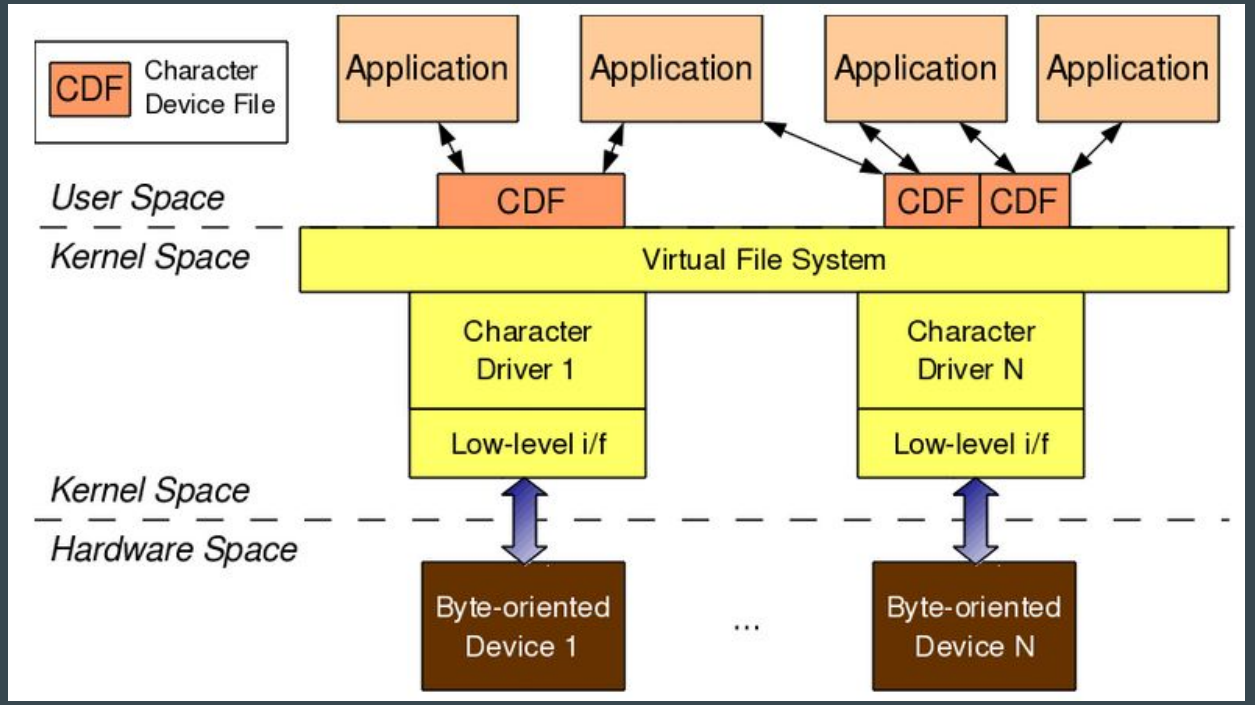
De hecho, todos los Device Driver que no son drivers de almacenamiento ni de red son algún tipo de CDD.

Veamos los puntos en común de estos CDD, las capas que intervienen y cómo empezar a escribir uno.

¿ Qué hay de especial en los CDD ?

Un modelo de capas..!!

- Application, Character device file (CDF), Character device driver (CDD), Character device



Nuestro primer CDD

Un hecho interesante sobre el kernel es que es una implementación OO en C. Y esto es tan profundo que podemos observar este paradigma, presente en nuestro primer driver.
Cualquier driver de Linux consta de un constructor y un destructor.
Se llama al constructor de un módulo cada vez que **`insmod`** logra cargar el módulo en el núcleo.
Se llama al destructor del módulo cada vez que **`rmmod`** logra descargar el módulo del núcleo.
Se implementan con dos funciones habituales en el driver con las macros `module_init()` y `module_exit()` incluidas en el encabezado de `module.h`.

Vamos a trabajar con drv1.c..!! compilar y mostrar la info.

Número mayor y menor

El vínculo entre APPLICATION y CDF se basa en el nombre del archivo del dispositivo. Sin embargo, el vínculo entre el CDF y DD se basa en el número del archivo de dispositivo. NO en el nombre.

Así una app del espacio de usuario tiene cualquier nombre para el CDF, y permite que el espacio del núcleo tenga, entre el CDF/CDD, un enlace trivial basado en un índice.

Este índice se conoce como el par `<major, minor>` del archivo del dispositivo. Lo puedo chequear en :

```
$ ls -l /dev/ | grep "^c"
$ cat /proc/devices
$ ls -la /dev/hda? /dev/ttyS?
```

Recursos para <mayor, menor> del kernel 2.6 en adelante

tipos especiales: (definido en la cabecera del kernel <linux/types.h>)

```
dev_t                   // contiene ambos números major & minor
```

macros específicas : (definidas en la cabecera del kernel <linux/kdev_t.h>)

```
MAJOR(dev_t dev)        // extrae el número mayor contenido en dev
MINOR(dev_t dev)        // extrae el número menor contenido en dev
MKDEV(int major, int minor) // crea dev  a partir de major & minor
```

Pasos para conectar CDF con CDD

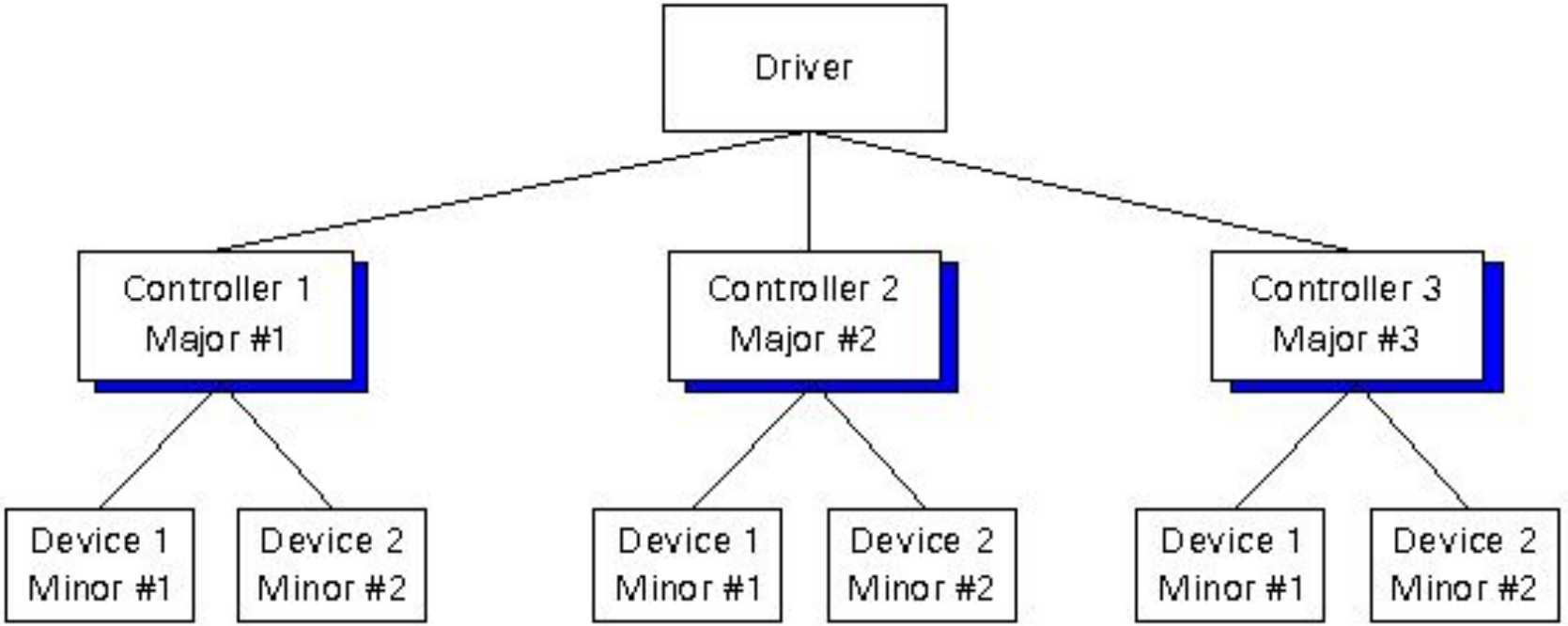
1. Registrar el rango <mayor, menor> para el CDD

Esto se logra utilizando cualquiera de estas dos API: (definidas en el encabezado del núcleo <linux/fs.h>)

```
int register_chrdev_region(dev_t from,
                           unsigned int cnt,
                           char *name);

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

Continuar con drv2.c..!!


Driver
Controller 1
Major #1
Controller 2
Major #2
Controller 3
Major #3
Device 1
Minor #1
Device 2
Minor #2
Device 1
Minor #1
Device 2
Minor #2
Device 1
Minor #1
Device 2
Minor #2

Registro de <major, minor>

- Compilamos, insertamos y verificamos que driver drv2 haya sido registrado.
- Verificamos el `major` asignado con: `cat /proc/devices`

  Pero no existe ningún archivo de dispositivo creado bajo `/dev` con el = número principal. Eso lo solucionamos creando los tres asociados con `mknod`

```
sudo mknod /dev/drv2_0 c MMM 0
sudo mknod /dev/drv2_1 c MMM 1
```

- Luego podemos intentar read/write esos CDF con

```
cat /dev/drv2_0
echo "Hola driver.." > /dev/drv2_0
```

Aparece un mensaje de error..!!

Creación automática de los CDF

Hasta el kernel 2.4, la creación de los archivos de dispositivo fue realizada automática por el kernel mismo, llamando a las API apropiadas de `devfs`.

A medida que el kernel evolucionó, los desarrolladores se dieron cuenta de que los DF son más una cuestión de espacio de usuario y, por lo tanto, como política, solo los usuarios deberían tratarlo, y no el kernel.

Con esta idea, ahora el núcleo solo completa, para el dispositivo en consideración, la clase de dispositivo y la información del dispositivo apropiadas en `/sys/class`

Luego, el espacio de usuario necesita interpretarlo y tomar una acción apropiada. En la mayoría de los sistemas de escritorio Linux, el demonio `udev` recoge esa información y, en consecuencia, crea los archivos de dispositivo.

Revisar los dispositivos de red en /sys

ls /sys/class/net/

enp0s25/ lo/ lxcbr0/ virbr0/ wlp3s0/

cat /sys/class/net/enp0s25/address

cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

¿Que otro sys interesante pueden encontrar ?

Pasos para conectar CDF con CDD

1. Registrar el rango <mayor, menor> para el CDD
2. Vincular las operaciones del CDF a las funciones del CDD.

Creación automática de los CDF

La clase de dispositivo se crea de la siguiente manera:

```
struct class *cl=class_create(THIS_MODULE, "<device class name>");
```

y luego la información del dispositivo (<major, minor>) se completa en esta clase con:

```
device_create(cl, NULL, first, NULL, "<device name format>", …);
```

donde `first` es tipo `dev_t` con el correspondiente `<major, minor>`.

Las llamadas complementarias o inversas correspondientes, que deben llamarse en orden cronológicamente inverso, son las siguientes:

```
device_destroy(cl, first);
```

```
class_destroy(cl);
```

Creación automática de los CDF

`Hallazgo 7E5(link)..!!`

El sistema de archivos `sysfs` es un pseudo-sistema de archivos que proporciona una interfaz a las estructuras de datos del kernel. (Más precisamente, los archivos y directorios en `sysfs` proporcionan una vista de las estructuras kobject definidas internamente dentro del kernel). Los archivos en `sysfs` brindan información sobre dispositivos, módulos del kernel, sistemas de archivos y otros componentes del kernel.

El sistema de archivos `sysfs` normalmente se monta en `/sys`. Normalmente, el sistema lo monta automáticamente.

Decodificando las operaciones del CDF

Trabajamos con drv3

- A partir de `cat` y `echo`, debemos encontrar una respuesta clara a las acciones que se desencadenan en cada caso, interpretando los mensajes en el log de eventos del kernel. Utilizar:

```
#cat /dev/SdeC_drv3
#echo “Hola driver..” > /dev/SdeC_drv3

No obtenemos ninguna salida..!!

#dmesg
```

Decodificando las operaciones del CDF

Trabajamos con drv3

- Se puede concluir que el valor de retorno de las funciones my_open() y my_close() son triviales. Pero no así read() y write() que devuelven ssize_t.
- En los encabezados del núcleo resulta ser una palabra con signo.
- Por lo tanto, puede devolver un número negativo (ERR). O un valor positivo.
- Que para read sería el número de bytes leídos.
- Para write sería el número de bytes escritos.

Interpretar correctamente read/write y los resultados en ambos casos..!!

Mejorando read() y write()

Trabajamos con drv4

- Como siempre, lo compilamos y hacemos las verificaciones de rutina.
- A partir de `cat` y `echo`, debemos observar los detalles de lo que ocurre.

```
cat /dev/SdeC_drv3
echo -n “H” > /dev/SdeC_drv3
```

Módulo Clipboard /proc ?

Para terminar trabajamos con clipboard.c

Compilarlo e insertar el módulo en el kernel. Luego ejecutar los siguientes comandos:

```
echo “Hola mundo...” > /proc/clipboard
cat /proc/clipboard
```

Se pueden observar algunas diferencias:

- Este es un módulo que crea una entrada en `/proc`, pero no es un CDD.
- Para las nuevas versiones del kernel, se recomienda utilizar la estructura proc_ops en lugar de file_operations (link)

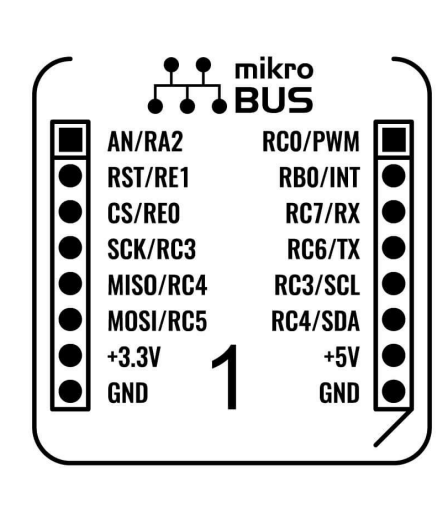
Algunas conclusiones

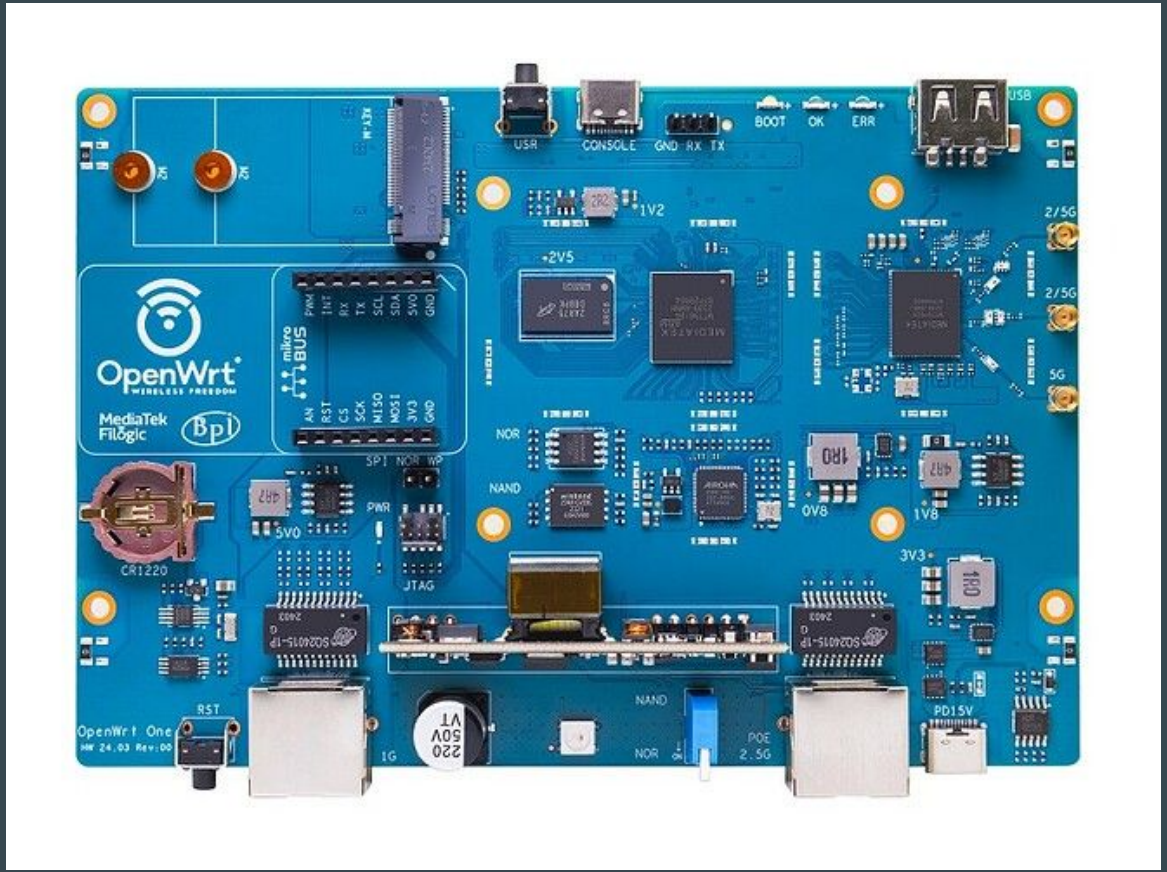
- Abordamos los detalles del proceso de construcción de CDD
- Queda unirlo a dispositivo de hardware que tendrá una serie de puertos GPIO mapeados en memoria
- Si trabajamos con Raspberry PI, hay material para completar el trabajo.

https://man7.org/linux/man-pages/man5/sysfs.5.html

https://man7.org/linux/man-pages/man5/proc.5.html

https://www.mikroe.com/gps-click





TP#5 cumplido...

más info ...

https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

para seguir indagando

.... Blocking I/O