微算機系統實習
MICROPROCESSOR SYSTEMS LAB.
SPRING, 2021

Instructor：Yen-Lin Chen(陳彥霖), Ph.D.
Professor
Dept. Computer Science and Information Engineering
National Taipei University of Technology

基本EMBEDDED LINUX 驅動程式設計

OUTLINE

- 設備驅動程式概論
- 編譯與執行驅動模組
- Char Device Driver 建置
- 與 I/O 進行溝通

設備驅動程式概論

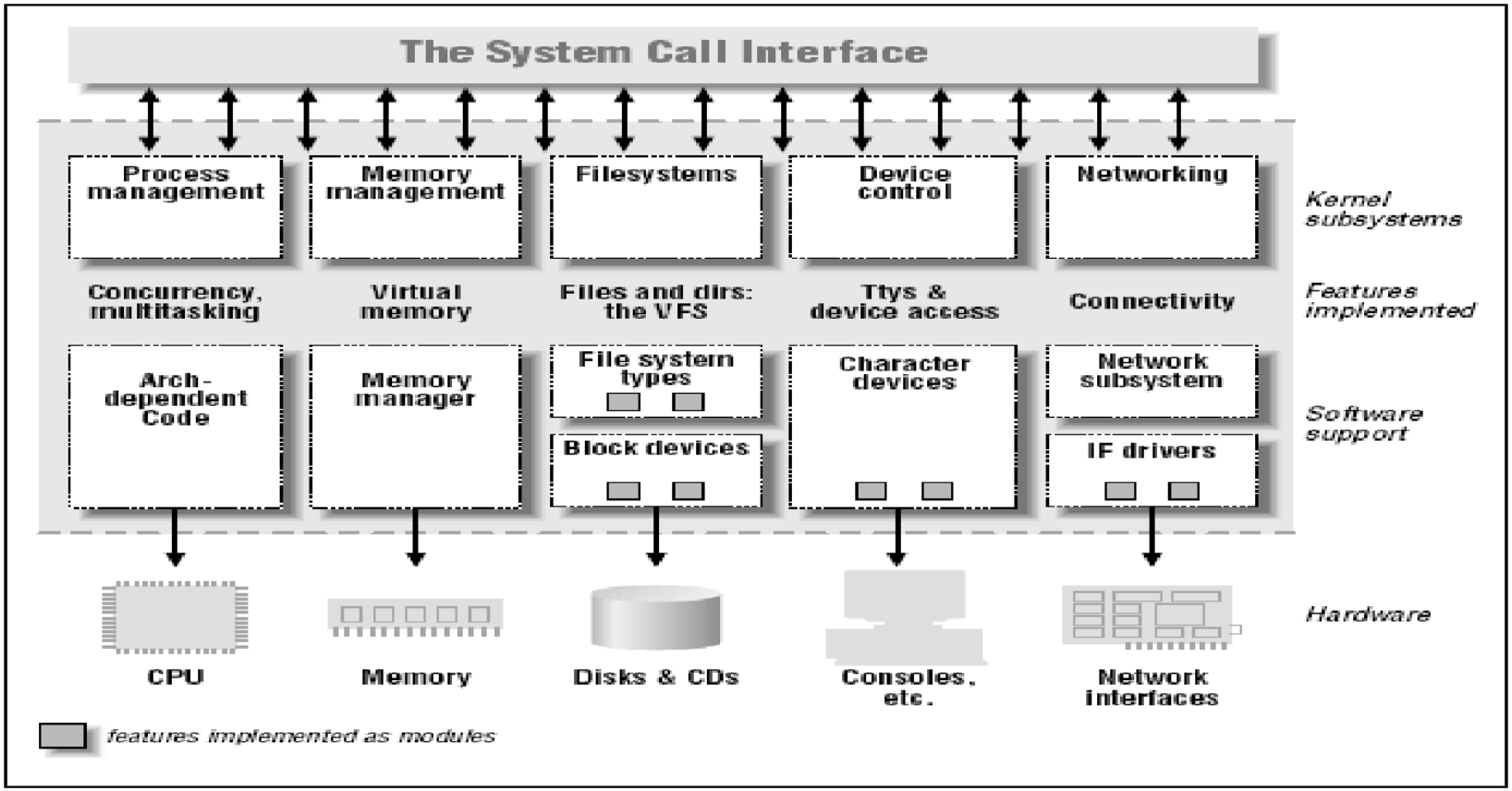
- The Role of the Device Driver
  - Black boxes.
  - User activities are performed by means of a set of standardized calls.
  - A software layer lies between the applications and the actual device.

設備驅動程式概論

- The architecture of the kernel
  - Process management
  - Memory management
  - File systems
  - Device control
  - Networking

設備驅動程式概論

- The architecture of the kernel

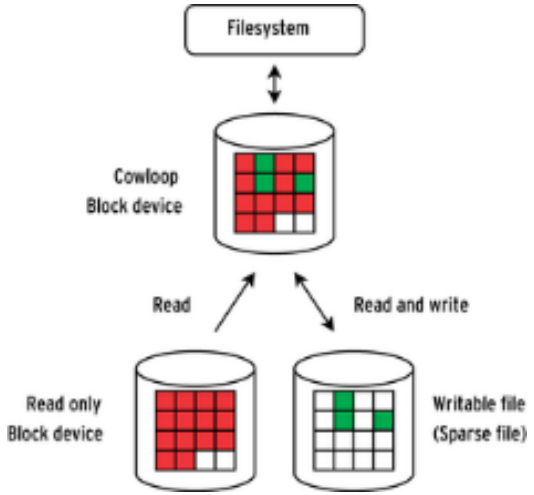


設備驅動程式概論

- Classes of Devices and Modules
  - Block devices
    - Like char devices, block devices are accessed by file system nodes in the /dev directory.
  - Network interfaces
    - Functions related to packet transmission.
  - Character devices
    - A character (char) device is one that can be accessed as a stream of bytes (like a file).

BLOCK DEVICES

- 區塊設備（block device）一種電腦裝置，如磁碟機，將數個字元組組成一個區塊，每次一個區塊的方式接收或傳送資料，而不是每次一個位元組（字元）的方式傳輸。

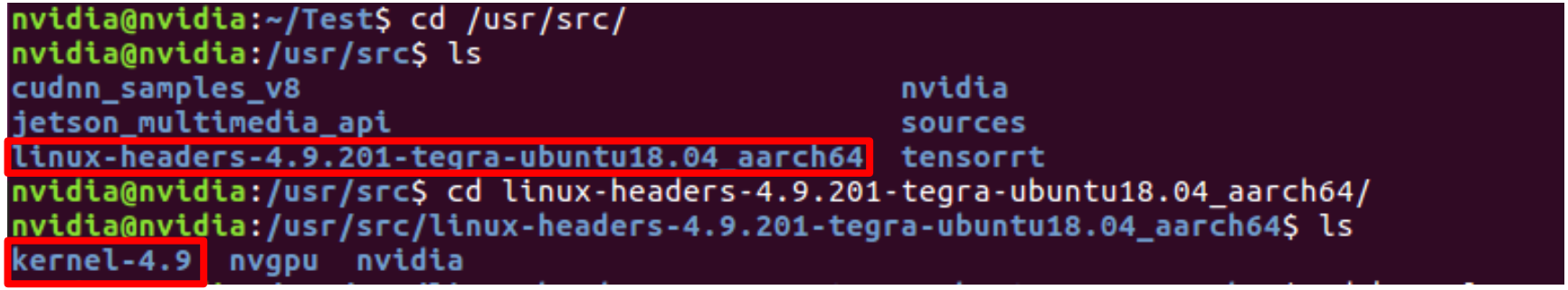


NETWORK INTERFACES

- 網路介面可以分為兩種形式存在，通常為具有實體的網路硬體設備，如乙太網卡或無線網卡等，而另外一種是以軟體來模擬硬體的網路介面，如loopback。在核心啟動時，系統通過網路設備驅動程式登記已經存在的網路設備。設備用標準的支援網路的機制來把收到的資料轉送到相應的網路層。

編譯與執行驅動模組

- 透過ssh連入嵌入式平台，查看/usr/src/目錄下是否有原先內建的kernel的資料夾(如下圖)



編譯與執行驅動模組

- 回到家目錄 (cd ~)或想放置編譯檔案的地方建立一個資料夾(ex: mkdir Test)並進入此資料夾(cd Test)

```
nvidia@nvidia:/usr/src$ cd ~
nvidia@nvidia:~$ mkdir Test
nvidia@nvidia:~$ cd Test
nvidia@nvidia:~/Test$
```

編譯與執行驅動模組

- 建立一個Makefile (gedit Makefile)內容如下:

```
obj-m := hellod.o
kernel_DIR := /usr/src/linux-headers-4.9.201-tegra-ubuntu18.04_aarch64/kernel-4.9/
PWD := $(shell pwd)
all:
	make -C $(kernel_DIR) SUBDIRS=$(PWD)
clean:
	rm *.o *.ko *.mod.c
.PHONY:
	clean
```

*此為輸出的驅動模組檔案名稱

*此為TX2 kernel資料夾位置

*此為編譯驅動模組

編譯與執行驅動模組

- Makefile內容(紅色部分會依照不同裝置或命名不同而更改)

```
obj-m := hellod.o

kernel_DIR := /usr/src/linux-headers-4.9.201-tegra-ubuntu18.04_aarch64/kernel-4.9/

PWD := $(shell pwd)

all: make -C $(kernel_DIR) SUBDIRS=$(PWD)

clean: rm *.o *.ko *.mod.c

.PHONY: clean
```

編譯與執行驅動模組

- 建立一個hellod.c內容如下:

Open hellod.c ~/Test

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

static int __init tx2_hello_module_init(void)
{
    printk("Hello, TX2 module is installed !\n");
    return 0;
}

static void __exit tx2_hello_module_cleanup(void)
{
    printk("Good-bye, TX2 module was removed!\n");
}

module_init(tx2_hello_module_init);
module_exit(tx2_hello_module_cleanup);
MODULE_LICENSE("GPL");
```

←掛載時會在系統紀錄檔印出此訊息

←卸載時會在系統紀錄檔印出此訊息

←此兩行設定驅動模組掛載與卸載時會呼叫哪個function

←設定此驅動模組的License（範例輸入為GPL）

編譯與執行驅動模組

- hellod.c驅動程式內容(紅色部分會依照不同裝置或命名不同而更改)

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

static int __init tx2_hello_module_init(void)
{
   printk("Hello, TX2 module is installed !\n");
   return 0;
}

static void __exit tx2_hello_module_cleanup(void)
{
   printk("Good-bye, TX2 module was removed!\n");
}

module_init(tx2_hello_module_init);
module_exit(tx2_hello_module_cleanup);
MODULE_LICENSE("GPL");
```

編譯與執行驅動模組

- 完成編輯上述程式後就可以開始編譯此驅動模組
  - make

```
nvidia@nvidia:~/Test$ ls
hellod.c  Makefile
nvidia@nvidia:~/Test$ make
make -C /usr/src/linux-headers-4.9.201-tegra-ubuntu18.04_aarch64/kernel-4.9/ SUBDIRS=/ho
me/nvidia/Test
make[1]: Entering directory '/usr/src/linux-headers-4.9.201-tegra-ubuntu18.04_aarch64/ke
rnel-4.9'
  LD      /home/nvidia/Test/built-in.o
  CC [M]  /home/nvidia/Test/hellod.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/nvidia/Test/hellod.mod.o
  LD [M]  /home/nvidia/Test/hellod.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.9.201-tegra-ubuntu18.04_aarch64/ker
nel-4.9'
nvidia@nvidia:~/Test$ ls
built-in.o  hellod.ko     hellod.mod.o  Makefile       Module.symvers
hellod.c    hellod.mod.c  hellod.o      modules.order
nvidia@nvidia:~/Test$
```

- 編譯出來的檔案中有一個hellod.ko此為驅動模組檔案

編譯與執行驅動模組

- 掛載與卸載驅動模組
  - insmod : loading module
  - rmmod : unload module
  - 搜尋系統上掛載的驅動模組是否有hellod
  - lsmod | grep hellod
- 查看系統訊息(可以查看模組掛載或卸載後printk的內容)
  - dmesg

編譯與執行驅動模組

- 掛載與卸載驅動模組

```
nvidia@nvidia:~/Test$ sudo insmod hellod.ko
nvidia@nvidia:~/Test$ sudo rmmod hellod.ko
```

- 查看系統訊息

```
nvidia@nvidia:~/Test$ dmesg
```

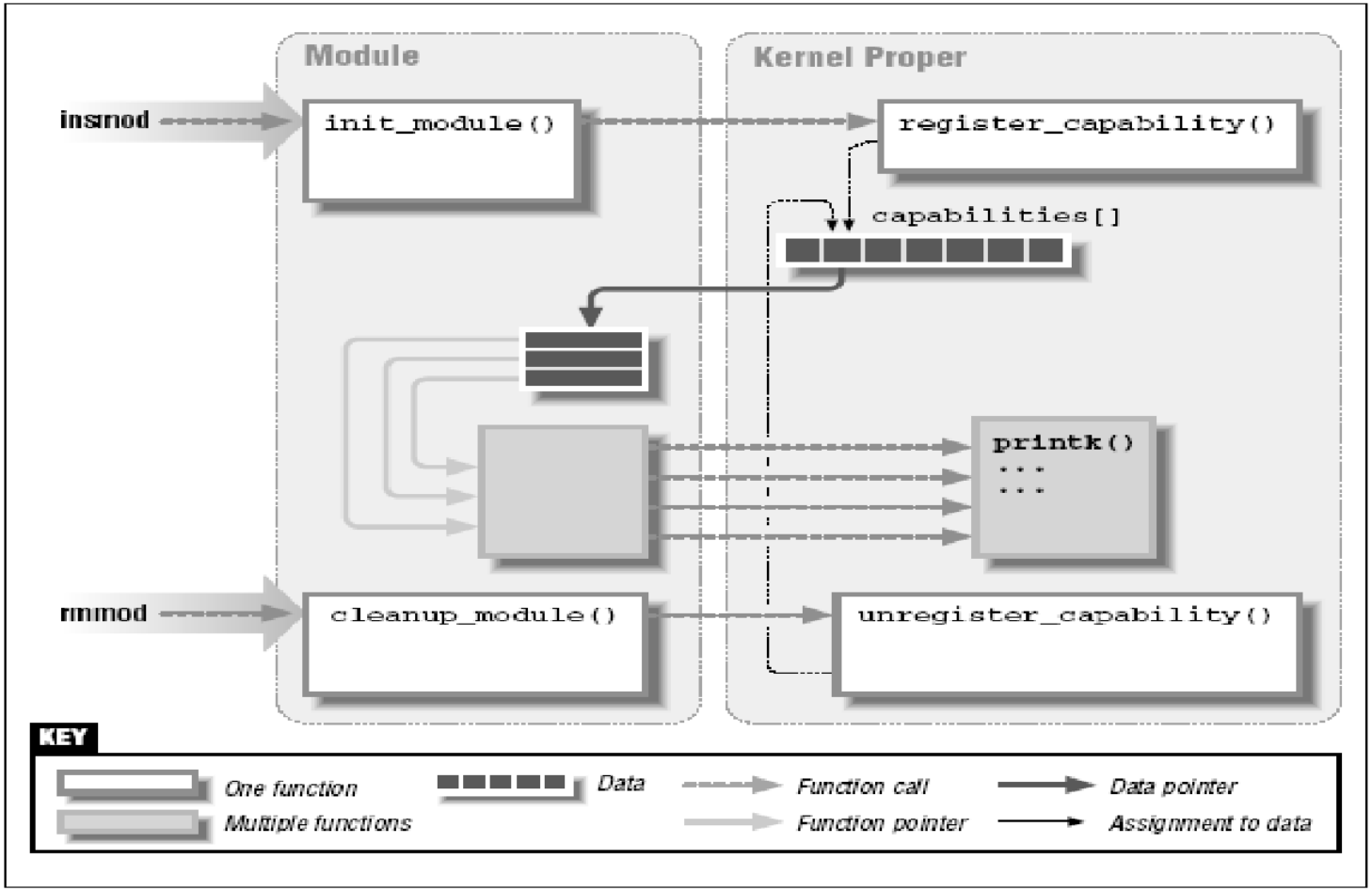
```
1413.352317] Hello, TX2 module is installed
1437.125210] Good-bye, TX2 module was removed!
```

- 掛載後查看系統掛載的驅動模組中是否有hellod

```
nvidia@nvidia:~/Test$ lsmod | grep hellod
hellod                  1341  0
```

編譯與執行驅動模組

- The relation of module and kernel



編譯與執行驅動模組

- User Space and kernel Space
  - The kernel executes in the highest level (also called supervisor mode)
  - Applications execute in the lowest level (the so called user mode)

編譯與執行驅動模組

- The Usage Count
  - Why to use?
    - The module can be safely removed.
  - To work with the usage count, use these three macros:
    - MOD_INC_USE_COUNT
      - Increments the count for the current module
    - MOD_DEC_USE_COUNT
      - Decrements the count
    - MOD_IN_USE
      - Evaluates to true if the count is not zero
  - The current value of the usage count is found in the third field of each entry in /proc/modules.

```
nvidia@nvidia:~$ cd /proc
nvidia@nvidia:/proc$ cat /proc/modules
hellod 1341 0 - Live 0x0000000000000000 (O)
bnep 16562 2 - Live 0x0000000000000000
fuse 103969 3 - Live 0x0000000000000000
```

編譯與執行驅動模組

- To unload a module
  - rmmod invokes delete_module that calls cleanup_module

BASIC DEVICE DRIVER

- 在前段落我們是在嵌入式平台(TX2)上編譯操作簡易的驅動，下一章節範例環境會以 X86 版本的 Ubuntu Linux(**虛擬機**) 來示範編譯操作，但實際操作過程基本上是相同的

BASIC DEVICE DRIVER
(UBUNTU LINUX X86)

- 製作編譯裝置驅動的 Makefile
  - 使用 uname –a 查看目前的 kernel版本
    - 因為每台虛擬機可能在更新後會更新Kernel版本，所以可能與下圖版本不同
  - 當知道kernel版本後再到/usr/src找到對應的kernel source

```
nvidia@ubuntu:~$ uname -a
Linux ubuntu 4.15.0-39-generic #42~16.04.1-Ubuntu SMP Wed Oct 24 17:09:54 UTC 20
18 x86_64 x86_64 x86_64 GNU/Linux
nvidia@ubuntu:~$ cd /usr/src/
nvidia@ubuntu:/usr/src$ ls
linux-headers-4.15.0-29          linux-headers-4.15.0-39
linux-headers-4.15.0-29-generic  linux-headers-4.15.0-39-generic
nvidia@ubuntu:/usr/src$
```

BASIC DEVICE DRIVER
(UBUNTU LINUX X86)

- 製作編譯裝置驅動的 Makefile
  - 找到kernel source位置後就可以開始編寫 Makefile

```
obj-m := hellod.o

kernel_DIR := /usr/src/linux-headers-4.15.0-39-generic/

PWD := $(shell pwd)

all:
	make -C $(kernel_DIR) SUBDIRS=$(PWD)
clean:
	rm *.o *.ko *.mod.c
.PHONY:
	clean
```

↑此處的kernel為Ubuntu 16.04版
不同於TX2嵌入式平台Ubuntu 18.04版

編譯與執行驅動模組
(UBUNTU LINUX X86)

- 完成編輯上述程式後就可以開始編譯此驅動模組
  - make

```
nvidia@ubuntu:~/Test_1$ ls
hellod.c  Makefile
nvidia@ubuntu:~/Test_1$ make
make -C /usr/src/linux-headers-4.15.0-39-generic/ SUBDIRS=/home/nvidia/Test_1
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-39-generic'
Makefile:975: "Cannot use CONFIG_STACK_VALIDATION=y, please install libelf-dev,
libelf-devel or elfutils-libelf-devel"
  CC [M]  /home/nvidia/Test_1/hellod.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/nvidia/Test_1/hellod.mod.o
  LD [M]  /home/nvidia/Test_1/hellod.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-39-generic'
nvidia@ubuntu:~/Test_1$ ls
hellod.c   hellod.mod.c  hellod.o  modules.order
hellod.ko  hellod.mod.o  Makefile  Module.symvers
nvidia@ubuntu:~/Test_1$
```

- 編譯出來的檔案中有一個hellod.ko此為驅動模組檔案

編譯與執行驅動模組
(UBUNTU LINUX X86)

- 掛載與卸載驅動模組
  - insmod : loading module
  - rmmod : unload module
  - 搜尋系統上掛載的驅動模組是否有hellod
  - lsmod | grep hellod
- 查看系統訊息(可以查看模組掛載或卸載後printk的內容)
  - dmesg

編譯與執行驅動模組
(UBUNTU LINUX X86)

- 掛載與卸載驅動模組

```
nvidia@ubuntu:~/Test_1$ sudo insmod hellod.ko
[sudo] password for nvidia:
nvidia@ubuntu:~/Test_1$ sudo rmmod hellod.ko
```

- 查看系統訊息

```
nvidia@ubuntu:~/Test_1$ dmesg
```

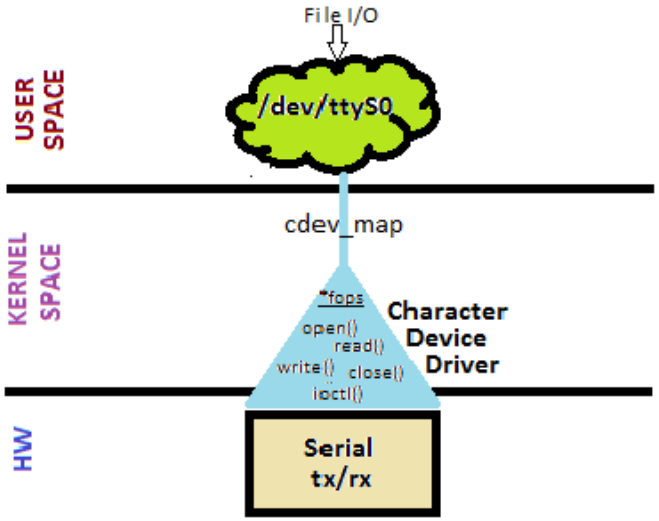
```
[ 2379.698089] Hello, Vitual Machine module is installed
[ 2422.194159] Good-bye, Vitual Machine module was removed!
```

- 掛載後查看系統掛載的驅動模組中是否有hellod

```
nvidia@ubuntu:~/Test_1$ sudo insmod hellod.ko
nvidia@ubuntu:~/Test_1$ lsmod | grep hellod
hellod                 16384  0
```

CHARACTER DEVICES

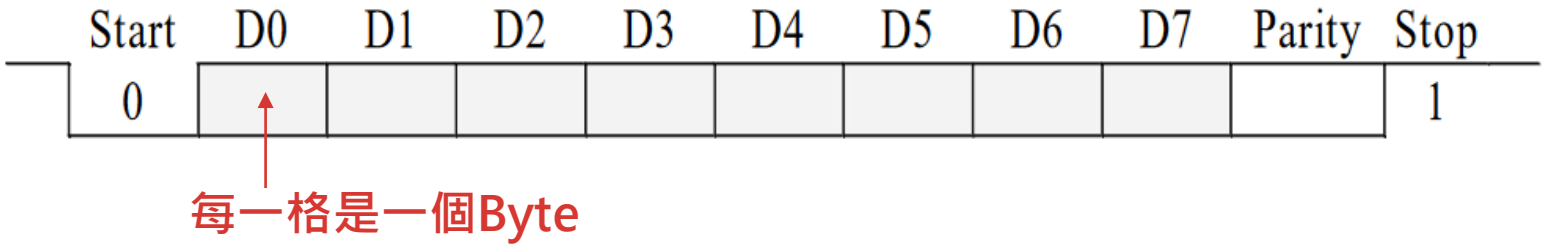
- 字元設備（character device）是一種電腦裝置，如鍵盤或列印機，每次一個字元的方式接收或傳送資料。字元的傳送可能是一個位元一個位元的串列傳輸或平行傳輸，但不是一次同時傳送數個位元組。



CHARACTER DEVICES

- Character Device是驅動程序通過發送和接收單個字元與通訊的設備。

  Character Device的範例：serial ports, parallel ports, sounds cards.



CHAR DEVICE DRIVER

- Make a device node in /dev（建立裝置檔案）
  - 我們需要先在/dev目錄下建立一個裝置檔案(node)，作為驅動模組掛載註冊時對應的裝置檔案
  - 建立裝置檔案(node)的方法
    - mknod /dev/〈裝置檔案名稱〉c〈主要版本號碼〉〈次要版本號碼〉
      - Ex: mknod /dev/demo c 60 0
        - demo 為裝置名稱
        - c 代表字元裝置
        - 60 為主要版本號碼
        - 0 為次要版本號碼
    - 提醒：建立裝置檔案必須要為root權限

CHAR DEVICE DRIVER

- Make a device node in /dev（建立裝置檔案）
  - sudo mknod /dev/demo c 60 0
  - 建立裝置檔案完成後可以在/dev目錄下看到新增的裝置檔案
  - 使用ls -l demo 可以查看名稱為demo這個裝置檔案相關內容

```
nvidia@ubuntu:/$ sudo mknod /dev/demo c 60 0
[sudo] password for nvidia:
nvidia@ubuntu:/$ cd /dev/
nvidia@ubuntu:/dev$ ls -l demo
crw-r--r-- 1 root root 60, 0 Apr 29 07:30 demo
```

此處的c代表Char Device Driver

CHAR DEVICE DRIVER

- 製作編譯裝置驅動的 Makefile
  - 在前段落中我們提到在嵌入式平台上要編譯裝置驅動所以要建立一個 Makefile，在 Makefile 要修改 kernel_DIR (kernel的source位置）與 obj-m（模組名稱），在 X86版本的Ubuntu Linux 也要做此件事情

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 在前段落中我們範例是簡單的驅動模組，他並沒有與系統註冊裝置檔案關聯，也沒有提供I/O的操作對應
  - 一般的字元裝置驅動存取會有open、close、I/O control、read 和 write 架構
  - 當我們像系統註冊上述這些架構的對應function後，這些function就如同是個別的event服務
  - 以下會介紹基本架構的function工作目的

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - Initial module
    - 當 driver 被載入之後第一個被呼叫的函式，類似一般 C 語言中的 main function，在此 function 中向系統註冊為字元 device 和所提供的服務
  - Open device
    - 當我們的 device 被 fopen 之類的函式開啟時所執行的對應處理函式
  - Close device
    - 使用者程式關閉我們的 device 時執行的對應處理函式

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - I/O control
    - 使用者可透過 ioctl 命令設定 device 的一些參數
    - 注意: 自Linux Kernel 2.6.35 把ioctl這個成員給移除了改用了以下兩名新成員
      - long (unlocked_ioctl) (struct file *, unsigned int, unsigned long);
      - long (compat_ioctl) (struct file *, unsigned int, unsigned long);
      - 我們一般情況下使用 unlocked_ioctl 來取代以前的 ioctl
  - Read device
    - 當程式從我們的 device 讀取資料時對應的處理函式

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - Write device
    - 當程式對我們的 device 寫入資料時對應的處理函式
  - Remove module
    - 當 driver 被移除時所執行的處理函式，必須對系統取消註冊 device

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組

*demo.c

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>

static int iCount = 0;
```

*宣告此iCount 我們等會兒在做write時候會將此變數+1，這樣我們可以觀察看看掛載裝置驅動後是否每次呼叫event服務時，變數的上一次內容是否依然存在

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 建立一個結構名稱為drv_fops將read、write、I/O control、open、release的function進行對應

```
struct file_operations drv_fops =
{
        read: drv_read,
        write: drv_write,
        unlocked_ioctl: drv_ioctl,
        open: drv_open,
        release: drv_release,
};
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 掛載裝置驅動時，在初始化我們要像系統註冊此模組對應的裝置檔案
    - register_chrdev(<主要版本號碼>, <裝置檔案名稱>, &<對應動作觸發function的結構>)

```
#define MAJOR_NUM 60
#define MODULE_NAME "demo"

static int demo_init(void)
{
        if (register_chrdev(MAJOR_NUM, "demo", &drv_fops) < 0)
        {
                printk("<1>%s: can't get major %d\n", MODULE_NAME, MAJOR_NUM);
                return (-EBUSY);
        }

        printk("<1>%s: started\n", MODULE_NAME);
        return 0;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - exit 的 function

```
static void demo_exit(void)
{
        unregister_chrdev(MAJOR_NUM, "demo");
        printk("<1>%s: removed\n", MODULE_NAME);
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 註冊模組初始化與結束對應的function

```
module_init(demo_init);
module_exit(demo_exit);
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - read 的 function
  - 參數中的char *buf 為傳入的內容，回傳的資料也必須寫在這裡(inout)
  - 參數中的size_t count 為傳入內容的大小

```
static ssize_t drv_read(struct file *filp, char *buf, size_t count, loff_t *ppos)
{
        printk("device read\n");
        return count;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - write 的 function
  - 參數中的char *buf 為傳入的內容
  - 參數中的size_t count 為傳入內容的大小
  - 在此function中我們印出iCount並對他+1，查看之後再次呼叫時是否變數會持續往上加
  - 我們也印出傳入的內容(buf)

```
static ssize_t drv_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
        printk("device write\n");
        printk("%d\n",iCount);
        printk("W_buf_size: %d\n",(int)count);
        printk("buf: %s\n",buf);
        iCount++;
        return count;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - ioctl 的 function

```
long drv_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    printk("device ioctl\n");
    return 0;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - open 的 function

```
static int drv_open(struct inode *inode, struct file *filp)
{
        printk("device open\n");
        return 0;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - release 的 function

```
static int drv_release(struct inode *inode, struct file *filp)
{
        printk("device close\n");
        return 0;
}
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 編寫一個控制/dev/demo的程式為test.o，範例程式碼如下

```c
#include <stdio.h>
int main()
{
        char buf[1024] = "Data Input 123456 hello world";
        FILE *fp = fopen("/dev/demo", "w+");
        if(fp == NULL) {
                printf("can't open device!\n");
                return 0;
        }
        fwrite(buf, sizeof(buf), 1, fp); //write
        fread(buf, sizeof(buf), 1, fp); //read
        fclose(fp);
        return 0;
}
```

補充：在新版的Ubuntu Linux 不建議程式直接使用ioctl來操作device，所以我們要控制字元裝置檔案必須透過開檔、讀檔與寫檔的方式來達成。

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - Make demo.ko 與 編譯test.c成test.o

```
nvidia@ubuntu:~/Test$ ls
demo.c  Makefile  test.c
nvidia@ubuntu:~/Test$ gcc -o test.o test.c
nvidia@ubuntu:~/Test$ ls
demo.c  Makefile  test.c  test.o
```

```
nvidia@ubuntu:~/Test$ make
make -C /usr/src/linux-headers-4.15.0-99-generic/ SUBDIRS=/home/nvidia/Test
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-99-generic'
  CC [M]  /home/nvidia/Test/demo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/nvidia/Test/demo.mod.o
  LD [M]  /home/nvidia/Test/demo.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-99-generic'
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - sudo insmod demo.ko 並查看dmesg

```
[ 3980.551382] <1>demo: started
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 執行數次test.o查看dmesg變化 (下面為連續執行兩次)

```
nvidia@ubuntu:~/Test$ sudo ./test.o
nvidia@ubuntu:~/Test$ sudo ./test.o
nvidia@ubuntu:~/Test$ dmesg
```

```
[ 4061.932503] device open
[ 4061.932507] device ioctl
[ 4061.932514] device write
[ 4061.932514] 0
[ 4061.932515] W_buf_size: 1024
[ 4061.932515] buf: Data Input 123456 hello world
[ 4061.932516] device read
[ 4061.932518] device close
[ 4064.055858] device open
[ 4064.055864] device ioctl
[ 4064.055870] device write
[ 4064.055870] 1
[ 4064.055871] W_buf_size: 1024
[ 4064.055871] buf: Data Input 123456 hello world
[ 4064.055872] device read
[ 4064.055874] device close
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - 用echo的方式將資料傳入/dev/demo裝置中
  - #echo “test123” > /dev/demo
  - (需要切換為root)

```
nvidia@ubuntu:~/Test$ sudo su
root@ubuntu:/home/nvidia/Test# echo "test123" >/dev/demo
```

```
[75553.529390] device open
[75553.529399] device write
[75553.529400] 7
[75553.529400] W_buf_size: 8
[75553.529402] buf: test123
```

CHAR DEVICE DRIVER

- 編寫完整的裝置驅動模組
  - sudo rmmod demo (demo.ko)

```
nvidia@ubuntu:~/Test$ sudo rmmod demo
```

```
[78605.174227] <1>demo: removed
```

dmesg訊息

掛載多個驅動注意事項

- 在/dev/下建立的node主要設備編號要不同
  - mknod /dev/node1 c 60 0
  - mknod /dev/node2 c 61 0
- 驅動程式撰寫時與系統註冊使用裝置的編號也要更改
  - register_chrdev(60, “node1", &drv_fops)
  - register_chrdev(61, “node2", &drv_fops)

```
#define MAJOR_NUM 60
#define MODULE_NAME "demo"

static int demo_init(void)
{
        if (register_chrdev(MAJOR_NUM, MODULE_NAME, &drv_fops) < 0)
        {
                printk("<1>%s: can't get major %d\n", MODULE_NAME, MAJOR_NUM);
                return (-EBUSY);
        }

        printk("<1>%s: started\n", MODULE_NAME);
        return 0;
}
```

EMBEDDED LINUX DRIVER 補充資料

OUTLINE

- 此段落主要補充更深入的Linux kernel函式庫與驅動程式編寫上與一般C/C++不同之處與要注意的地方
  - 使用 Linux kernel 的函式庫進行檔案處理
  - 掛載多個驅動注意事項

使用 LINUX KERNEL 的函式庫進行檔案處理

- 檔案處理分為下列四項
  - 開檔
  - 讀檔
  - 寫檔
  - 關檔
- 使用kernel library做檔案處理必須include linux/fs.h、asm/uaccess.h

讀取 USER SPACE 的資料

- 我們是不能直接存取 buf 的內容。因為 Kernel Space 與 User Space 有不同的位址空間，所以不能直接存取他們。我們必須借助 copy_from_user 這個 API。
- 必須先include <asm/uaccess.h>

```
static ssize_t drv_write(struct file *filp, const char *buf, size_t count, loff_t *ppos){
        printk("Enter Write function\n");
        printk("%d\n", iCount);
        printk("W_buf_size: %d\n", (int)count);

        copy_from_user(userChar, buf, count);
        userChar[count - 1] = 0;

        printk("userChar(chr): %s\n", userChar);
        printk("userChar(int): %d\n", (int)sizeof(userChar));
```

userChar 資料型態為char array

使用 LINUX KERNEL 的函式庫進行檔案處理

- 開檔
  - 宣告檔案結構指標
    - struct file *fp;
  - 開啟檔案
    - fp = filp_open("<檔案路徑>", <開檔方式>, <權限>);
    - 檔案路徑：要開啟檔案的路徑
    - 開檔方式：有下列三種參數，若要同時使用２個（含）以上的方式可以使用OR(|)的方式來填寫，例如要使用讀寫檔案且當檔案沒有時候要創建就要輸入O_RDWR| O_CREAT
      - O_RDWR：Open for reading and writing. The result is undefined if this flag is applied to a FIFO.
      - O_WRONLY ：Open for writing only.
      - O_RDONLY：Open for reading only.
      - O_CREAT：若沒有檔案時創建檔案
    - 權限：Linux檔案權限（４為數字），預設設０即可

使用 Linux kernel 的函式庫進行檔案處理

- 開檔前要先將記憶體資料存起來當檔案處理完關檔案後要寫回記憶體
- mm_segment_t old_fs; //建立紀錄記憶體中資料的結構
- old_fs = get_fs(); //取得記憶體中資料放
- set_fs(get_ds()); //將記憶體寫檔區段清空讓之後寫檔使用
- ～～～寫檔並關檔後～～～
- set_fs(old_fs); //回存記憶體中資料區段

使用 LINUX KERNEL 的函式庫進行檔案處理

- 關檔
- filp_close(<檔案結構指標>, NULL);

使用 LINUX KERNEL 的函式庫進行檔案處理

- 寫檔
- loff_t pos = 0; //從第幾個char開始寫
- vfs_write(<檔案結構指標>, <資料>, <資料大小>, &pos);

使用 LINUX KERNEL 的函式庫進行檔案處理

- 讀檔
- loff_t pos = 0; //從第幾個char開始讀
- vfs_read(<檔案結構指標>, <資料存放陣列>, <資料大小>, &pos);

使用 LINUX KERNEL 的函式庫進行檔案處理

```
struct file *fp; //建立檔案的結構
loff_t pos = 0; //從第幾個char開始寫
mm_segment_t old_fs; //記錄記憶體內的檔案資料
char buff[10] = { "1234567" };
old_fs = get_fs(); //記錄記憶體內的檔案資料
set_fs(get_ds());

fp = filp_open("/tmp/test.txt", O_WRONLY, 0);

printk("file is open!!\n");

vfs_write(fp, buff, strlen(buff), &pos);//寫檔
vfs_read(fp, buff, strlen(buff), &pos);//讀檔

filp_close(fp, NULL);//關檔
set_fs(old_fs); //之前資料寫回記憶體
```

參考資料

- http://pubs.opengroup.org/onlinepubs/7908799/xsh/open.html
- https://www.kernel.org/doc/html/v4.12/core-api/kernel-api.html