Raspberry Pi3
Linux Driver I 實驗模組建置

實驗模組名稱： Linux Driver (I)

開發學生： 溫方志、陳鎮國

開發教師： 陳鵬升

學校系所： 中正大學資訊工程學系

聯絡電話： 05-2720411 ext.33117

聯絡地址： 62102 嘉義縣民雄鄉大學路 168 號資訊工程學系

實驗平台： Ubuntu

實驗內容關鍵字：

- 實驗目的：

1. 寫一個 Driver 可以讓 kernel 掛載和卸載
2. 了解 Driver 的開發過程和運作

- 實驗器材：

1. Raspberry Pi3

- 實驗所需資源：

1. Cross Compiler
2. Kernel source code

目錄

Part 1 –module 的編譯和測試

Linux 的優點之一，可以在執行期(系統已啟動且正在運作)擴充核心的功能，且在用不到的時候卸載。可在執行期擴充的程式碼稱為模組(module)，模組有很多類型，驅動程式只是其中一種。模組是以 object code 的形式存在，因為要連結的對象都在核心中(所以編譯的時候需要 kernel source code 的相關檔案)，使用 insmod 可以把模組動態連結(載入)到正在運作的核心，反之，rmmod 可以把模組從核心移除(卸載)。

Step 1: Hello-World 模組的形態

<說明>
一個最簡單的 Linux Driver 形態

<實驗步驟>
1. hello.c

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

MODULE_LICENSE("Dual BSD/GPL");

static int demo_init(void) {
    printk("<1>I am the initial function!\n");
    return 0;
}

static void demo_exit(void) {
    printk("<1>I am the exit function!\n");
}

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

由於 driver 是在 kernel space 執行，平時我們常用的 system call 和 C function 都不能使用(Ex: printf)。上面程式所看到的 printk 就是在 kernel space 中功能和 printf 功能相近的函式。另外 init function 和 exit function 是可以任意命名的，只要使用

module_init 和 module_exit 巨集宣告即可。當我們執行 insmod 指令載入 driver 時，driver 中的 initial function 就會被呼叫，同樣的 exit function 會在執行 rmmod 指令時被呼叫。請大家 compile 這個範例並且使用 insmod 和 rmmod 測試一次。printk 的輸出一般來說會直接輸出在 console，如果沒有辦法看到任何輸出，請使用 dmesg 命令或者是到/var/log/syslog 中察看

Step 2: module 的 makefile

<說明>

PATH 是放置 Linux kernel source code 的路徑，-C 參數是會切換到[PATH]這個工作目錄，因為要編譯的是核心的模組，必須要使用核心的 source code，而 M=則會使 makefile 到目前的工作目錄建構 modules(即.ko 檔)。

<實驗步驟>

1. makefile 如下

```
obj-m := hello.o
all:
    make -C [kernel-source-code's PATH] M=$(PWD) modules
clean:
    make -C [kernel-source-code's PATH] M=$(PWD) clean
```

2. 使用此 makefile 編譯 Hello-World 模組

   % make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

Step 3: 測試 Hello, world 模組

<說明>

make 完之後你就可以得到 hello.ko 這個 module 檔，切換為 root 身份，你就可以使用 insmod 和 rmmod 來掛載和卸載。在 Ubuntu 裡，printk 的輸出要用 dmesg 這個指令才看得到。

<實驗步驟>

1. 使用 insmod 和 rmmod 來掛載和卸載模組
2. 用 dmesg 指令查看模組是否有掛載成功

Part 2 –完整 Driver 的形態

Linux 將 driver 分為三種型態，分別是字元、區塊(例如:磁碟)和網路設備(例如:網路卡)，我們以最簡單的字元裝置為範例，將上面的 hello world 模組加入基本的 open、close、I/O control、read、和 write，就是一個簡單有完整功能的 driver。在 driver 的基本架構中，我們首先向系統註冊一個 driver，再向系統註冊我們所提供的 open、close、read 和 write 的服務即可。

Step 1: Driver 架構

<說明>
完整的 Driver 架構
<實驗步驟>

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

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

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

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

```
long drv_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg) //2.6.36 version modify
{
   printk("device ioctl\n");
   return 0;
}

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

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

#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;
}
```

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

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

- 使用一個 struct file_operations 來設定所有操作對應的 function，這個 structure 的定義可以在 linux/fs.h 中找到。
- 在 initial module 的 function 中，透過 register_chrdrv 函式來註冊一個字元裝置，並將剛剛所設定的 structure 傳給系統。使用者便可透過一般的檔案操作函式來存取該 device，只要在 open、close、I/O control、read 和 write 等函式中加上對應的硬體操作，就可以完成一個簡單的 driver 了。
- 在 remove module 的時候必須呼叫 unregister_chrdev 函式取消裝置註冊，以免系統產生異常。
- Linux 內核到 2.6.36 之後把 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);

Step 2: 使用 Part1 的 makefile 編譯

<說明>
使用 Part1 的 makefile 編譯 Driver
<實驗步驟>
1. 使用 Part1 的 makefile 編譯此模組，產生 ko 檔

Step 3: 建立裝置節點

<說明>
在 Linux 底下所有的裝置都以檔案來代表，但是檔案如何代表該裝置呢？ 就是透過檔案的 major 與 minor 數值來替代的，通常只要 major 一樣都是用同一個驅動程式驅動，在早期的 Linux 版本 major num 可以代表某一個裝置，但是 Linux 的開發團隊極力淡化這樣的關係，甚至 major 一樣可能也用不同的 driver 去驅動，不過大部分的開發者還是默認早期的使用方式。

<實驗步驟>
需切換為 root 身份。
使用指令 mknod /dev/demo c 60 0
其中/dev/demo 是裝置名稱，c 代表字元裝置，60 代表 major，0 代表 minor，來建立我們的裝置結點

- 字元裝置 (character device)：字元裝置可以讓應用程式像操作檔案一樣來存取裝置，每次傳輸一個 byte，通常這類裝置至少需要實作 open、close、read、write 四個系統呼叫。相對而言區塊裝置 (block device)，是指可以儲存資料的裝置，例如：硬碟、光碟機等等，每次傳輸整個區塊的資料，區塊的大小通常是 512 bytes、或更大的 2 的次方大小。

Step 4: 寫一個程式測試 Driver 的讀寫

<說明>
當 driver 完成了之後，我們就可以寫一個簡單的測試程式來檢驗 driver 是否正常運作，其實方式也相當簡單，只要將我們剛剛建立的裝置檔案/dev/demo 當作一般檔案開啟並測試我們所寫的功能如 read、write 即可：

<實驗步驟>
1. 用 insmod 把 driver 掛載上去
2. test.c 之程式碼如下

```
#include <stdio.h>

int main()
{
    char buf[512];
    FILE *fp = fopen("/dev/demo", "w+");
    if (fp == NULL) {
        printf("cannot open device!\n");
        return 0;
    }
    fread(buf, sizeof(buf), 1, fp);
    fwrite(buf, sizeof(buf), 1, fp);
    fclose(fp);
    return 0;
}
```

3. 編譯此測試程式，使用: arm-linux-gnueabihf-gcc -static -g test.c，並把它放到實驗板上執行後，可以檢視 driver 的訊息輸出結果，觀察程式和 driver 通訊的流程(driver 的輸出可能由螢幕輸出，或者使用 dmesg、檢視 /var/log/syslog 等方式得到)。

Reference:

1. Linux Device Driver,3e
2. 嵌入式軟體聯盟之 PXA270 教材
3. “Writing a Linux character Device Driver”, https://appusajeev.wordpress.com/2011/06/18/writing-a-linux-character-device-driver/.