## <http://www.jollen.org/LinuxDeviceDriver/>

## Linux 驅動程式觀念解析, #1: 驅動程式的大架構

**Linux 驅動程式的大架構**

Linux 驅動程式的整體架構如下：

1. application 透過 system call 介面與 kernel 溝通。
2. 透過 kernel 的 VFS 層與 Linux 驅動程式物件溝通。

**Linux 驅動程式三類型**

Linux device driver 可分成 3 種類型：character device driver, block device driver, network device driver。

驅動程式本身可分成 2 個層面來討論：virtual device driver, physical device driver。

**Virtual Device Driver**

往上層支援Linux kernel所提供的Virtual File System層，並藉此實作system calls。使用者可透過 system call interface 與device driver溝通。Virtual device driver的重要性大於physical device driver，如何善用Linux所提供的介面(interface)來設計驅動程式，並配合user application來設計應用程式是重點。Virtual device driver的目的在於善用Linux的APIs來設計機制(mechanism)與行為 (behavior)良好的驅動程式，因此「觀念」的重要性遠大於「語法」的討論。

**Physical Device Driver**

往下層使用Linux kernel所提供的device interface來存取並控制實體硬體裝置。Physical device driver 則是討論「如何透過I/O port或I/O memory」來控制裝置，也就是與晶片組的溝通。這個部份需要實作晶片組的 data sheet。

**小結**

OS 是設計良好的軟硬體介面，Linux 驅動程式設計即是在學習如何使用 Linux 提供的介面來設計驅動程式。在OS上應使用設計良好的API來撰寫驅動程式，使用OS API的優點是使驅動程式設計抽象化(abstraction)，可不需要太深入硬體層次。

Linux kernel 所提供的 API 均經過良好設計，因此使用 kernel所提供的 API 可以確保系統運行的安全性與穩定性。

## Linux 驅動程式觀念解析, #2: System Calls

**System Call 與驅動程式的關係**

System call是user application與Linux device driver的溝通介面。User application 透過呼叫 system call來「叫起」driver的 task，user application要呼叫system call必須呼叫GNU C所提供的「wrapper function」，每個 system call 都會對應到 driver 內的一個 task，此 task 即是 file\_operation 函數指標所指的函數。

Linux 驅動程式與user application間的溝通方式是透過 system call，實際上user application是以device file與裝置驅動程式溝通。要達成此目的，驅動程式必須建構在此「file」之上，因此Linux驅動程式必須透過VFS（virtual file system）層來實作system call。

**一個簡單的範例**

/dev目錄下的檔案稱為device file，是user application用來與硬體裝置溝通的介面。以下是一個簡單的範例：

*int main(int argc, char \*argv[]) {  
 int devfd;  
 devfd = open("/dev/debug", O\_RDONLY);  
 ioctl(devfd, IOCTL\_WRITE, num);  
 close(devfd);  
 return 0;  
}*

打開/dev/debug檔案時，範例所呼叫open()函數會叫起支援/dev/debug驅動程式的對應函數；同理，我們對/dev/debug執行ioctl()函數時，也會叫起驅動程式的相對應函數。範例中的open()與ioctl()函數皆是GLIBC裡的函數，「叫起」驅動程式函數的動作涉及user space與kernel space的切換，此動作藉由system call介面來完成。設計一個支援”/dev/debug”裝置的驅動程式則是Linux驅動程式設計師所要負責的工作。

## Linux 驅動程式觀念解析, #3: Device File

**什麼是 Device File**

在UNIX系統底下把外部的周邊裝置均視為一個檔案，並透過此檔案與實體硬體溝通，這樣的檔案就叫做device files。Linux device driver與user的重要溝通橋梁為device files。檔案屬性的第一個位元如果顯示為 “c” 表示這是一個字元型裝置的 device file、若為 “b” 表示這是一個區塊型裝置的Device file。

Device file的major number表一個特定的裝置，例如major number為1為null虛擬裝置，major number 定義於 kernel文件目錄Documentation/devices.txt 。Minor number代表裝置上的子裝置，例如同一個硬碟上的分割區就用不同的minor number來代表，但其major number相同。

**Device File 與驅動程式的關係**

在設計device driver時，會先透過一個“註冊” (register)的動作將自己註冊到kernel裡，註冊時會指定一個 major number 參數，以指定此驅動程式所要實作的週邊裝置。當user開啟device file時，kernel便會根據device file的major number 找到對應的驅動程式來回應使用者。Minor number則是 device driver內部所使用，kernel 不會處理不同的 minor number。

設計device driver的第一個步驟就是要定義 driver所要提供的功能 (capabilities)，當 user application 呼叫open() system call 時，kernel 就會連繫相對應的 driver 來回應使用者。file\_operations是學習 device driver 最重要的一個資料結構，file\_operations內的成員為函數指標，指向“system call的實作函數”。file\_operations即是圖中的VFS層。換句話說，Linux驅動程式是透過file\_operations來建構VFS層的支援。而file\_operation裡的函數指標，即是指向每一個system call的實作函數。

## Linux 驅動程式觀念解析, #4: Linux 驅動程式一般化設計流程

**一般化設計流程**

對Linux驅動程式而言，virtual device driver的重要性在physical device driver之上，雖然沒有physical device driver無法真正驅動硬體。但實作上physical device driver是一成不變的寫法，好的驅動程式關鍵是在virtual device driver。

**struct file\_operations**

struct file\_operations是kernel提供的一個資料結構。Linux驅動程式建構在file\_operations之上，定義驅動程式的system call與實作system call的函數，file\_operations任何一個部份都能切成virtual device driver與physical device driver二個部份。

**流程解說**

Virtual device driver往上是連結Linux kernel的VFS層，physical device drvier往下是為了存取實體硬體。

**Virtual Device Driver**

virtual device driver也必須考慮與user application的互動。實作上，則是需要善用kernel所提供的介面（interface），即kernel APIs。Virtual device driver再分為3階段的觀念實作：1. 定義 file\_operations。2. 實作 system calls。3. 註冊 driver (VFS)。fops 是指向file\_operations結構的指標，驅動程式呼叫register\_chrdev()將fops註冊到kernel裡後，fops便成為該device driver所實作的system call進入點。實作system call的函數便是透過file\_operations結構來定義，我們稱實作system call的函數為driver method。

kernel會在需要時回呼(callback)我們所註冊的driver method。因此當driver裡的method被呼叫時，kernel便將傳遞參數(parameters)給driver method，driver method可由kernel所傳遞進來的參數取得驅動程式資訊。註冊driver的動作呼叫register\_chrdev()函數完成，此函數接受3個參數如下：1. major：要註冊的裝置 major number。2. name：device 名稱。3. fops：driver 的 file operation。「註冊」這個動作觀念上是將fops加到kernel的VFS層，因此user application必須透過「device file」才能呼叫到driver method。註冊這個動作的另一層涵意則是將driver method與不同的system call做「正確的對應」，當user application呼叫system call時，才能執行正確的driver method。

**Physical Device Driver**

Physical device driver的目的在於實作控制硬體的程式碼。Physical device driver的設計必須隨時查閱晶片(chipsets)的data sheet，並透過晶片的control register來控制裝置。理論上，我們可以將晶片的暫存器分成3大類：1. data registers。2. control registers。3. status registers。Data register是晶片裡用來存放資料的暫存器，control register則是用來控制晶片行為的暫存器，status register則保存目前晶片的狀態。設計控制硬體周邊的驅動程式時，需要了解硬體使用的晶片組，晶片組則需要參考IC設計廠商所提供的「datasheet」才能了解晶片組的暫存器名稱與用途，通常不同的暫存器會對應到一個「相對」的偏移位址（offset）。

驅動程式則是要透過control register才能控制晶片，因此需要隨時查閱晶片的datasheet，並了解每一個暫存器的用途。通常暫存器的每個位元(bit)也都是有特定用途的，因此設計驅動程式時，必須要很熟悉C語言的位元運算用法。實作上，首先會將晶片的datasheet寫成C語言的標頭檔，通常這個檔案都可以從 vendor 取得。接著再定義一組操作暫存器的I/O函數，我們稱這組函數為I/O wrapper function。I/O wrapper functions通常是重新定義Linux kernel所提供的readb()、writeb()或inb()、outb()系列函數所寫成的。最後，利用I/O wrapper function實作一系列的控制函數，以控制實際硬體，我們稱此函數為chipset control functions。Chipset control functions是由實作system calls的函數(driver method)所呼叫，因此在設計chipset control functions時也會回頭改寫driver method以符合此階段的實作。

## Linux 驅動程式觀念解析, #5: 依流程來實作 -- Virtual Device Driver

**根據流程寫程式**

**定義 file\_operations**

*struct file\_operations card\_fops = {*

*open: card\_open,*

*write: card\_write,*

*release: card\_release,*

*ioctl: card\_ioctl,*

*};*

由此定義驅動程式將提供open/write/close(release)/ioctl 4個system call介面給user application。

**實作 System Call**

接著要實作所提供的4個 system call。

**註冊 Driver**

將driver自己「註冊」到kernel的VFS層，註冊時所要呼叫的函數根據裝置類型的不同而不同。將驅動程式「註冊」(registration)至kernel的動作須在init\_module()函數裡實作。根據裝置類型的不同，所呼叫的函數也不同，以下是幾個裝置註冊函數：

˙ int register\_chrdev(unsigned int major, const char \* name, struct file\_operations \*fops)：註冊字元型驅動程式。  
˙ int register\_blkdev(unsigned int major, const char \*name, struct file\_operations \*fops)：註冊區塊型驅動程式。  
˙ int usb\_register(struct usb\_driver \*new\_driver)：註冊USB驅動程式。  
˙ int pci\_register\_driver(struct pci\_driver \*)：註冊PCI驅動程式。

本文範例註冊驅動程式的程式片斷如下：

*#define DEV\_MAJOR 121*

*#define DEV\_NAME "debug"*

*#define MSG(format, arg...) printk(KERN\_INFO "DEBUG CARD: " format "\n", ## arg)*

*int init\_module(void){*

*MSG("DEBUG CARD v0.1.1");*

*if (register\_chrdev(DEV\_MAJOR, DEV\_NAME, &card\_fops) < 0) {*

*MSG("Couldn't register a device.");*

*return -1;*

*}*

*return 0;*

*}*

register\_chrdev()參數說明如下：

˙ 第1個參數：為device file的major number。該device file應在Linux系統底下以root身份手動建立。  
˙ 第2個參數：  
˙ 第3個參數：為驅動程式的fops。

註冊的動作是在init\_module()裡，當使用者執行insmod載入驅動程式時，register\_chrdev()便會執行。因此註冊驅動程式的時機為insmod時。相對的在rmmod時，必須執行解除註冊的動作，此動作必須實作在cleanup\_module()函數裡。

前面所介紹的4個註冊函數，其相對應的解除註冊函數如下：

˙ int unregister\_chrdev(unsigned int major, const char \* name) ：解除註冊字元型驅動程式。  
˙ int unregister\_blkdev(unsigned int major, const char \*name) ：解除註冊區塊型驅動程式。  
˙ void usb\_deregister(struct usb\_driver \*driver)：解除註冊USB驅動程式。  
˙ pci\_unregister\_driver(struct pci\_driver \*drv) ：解除註冊PCI驅動程式。

範例debug card 0.1.0解除註冊的程式片斷如下：

*void cleanup\_module(void) {*

*if (unregister\_chrdev(DEV\_MAJOR, DEV\_NAME))*

*MSG("failed to unregister driver");*

*else*

*MSG("driver un-installed\n");*

*}*

Linux驅動程式的「註冊」是一個非常重要的動作，這個動作代表 Linux 驅動程式是一個嚴謹的分層式架構；換句話說，  
Linux驅動程式的分層（layered）關係可透過「註冊」的程序來分析。

**定義chipset標頭檔**

我們所要設計的 Port 80H 除錯卡驅動程式，不需要定義標頭檔；此部份可參考 kernel 裡的 BTTV 驅動程式。

**定義I/O wrapper function**

要設計的Port 80H除錯卡驅動程式，不需要定義 I/O wrapper function；此部份可參考 kernel 裡的 BTTV 驅動程式。

**實作chipset控制函數**

在要設計的 Port 80H 除錯卡驅動程式中，我們是直接使用 kernel 的 I/O 介面來控制除錯卡，physiacl device driver 的部份將在下一篇文章再做說明。

**open/release實作**

open與release是Linux驅動程式最基本的2個system call。驅動程式應先實作此2個system call。

open與release system call的執行時機如下：

1. 當user application執行open()函數時，便呼叫Linux kernel的open system call，即執行fops->open。

2. 當user application執行close()函數時，便呼叫Linux kernel的close system call，即執行fops->release。

Linux驅動程式註冊至kernel時會指定device file的major number，user application便可以透過此符合此major number的device file與硬體溝通，即Linux驅動程式是透過VFS架構層與user application溝通。

file\_operation是Linux驅動程式支援VFS的重要結構。學習file\_operation的重要目的如下：

1. 了解每一個system call的用途。

2. 了解每一個system call的實作「原則」。

System call的實作原則即driver function所要負責處理的基本工作。學習Linux device driver的重要工作之一，便是一一了解fops裡每一個system call的實作原則，並依照實際需求來實作不同的 system call。

**open System Call**

以open system call為例，fops->open是在user呼叫open()函數時執行，即當user開啟driver所指定的device file時呼叫fops->open。

fops->open實作原則如下：1. 將usage count加一(increment)。2. 檢查inode->i\_rdev。3. 檢查裝置是否錯誤。4. 初始化裝置。5. 將驅動程式自己的資料結構放到filp->private\_data。以下是本範例的fops->open實作：

int card\_open(struct inode \*inode, struct file \*filp) {

MOD\_INC\_USE\_COUNT;

return 0;

};  
**release System Call**

當user application 呼叫close() 函數後，便執行fops->release。有些驅動程式會將release method函數名稱命名為 XXX\_close()，但建議以XXX\_release()名稱為主，以避免混淆。fops->release實作原則如下：將 usage count 減一。

以下是本範例的fops->release實作：

int card\_release(struct inode \*inode, struct file \*filp) {

MOD\_DEC\_USE\_COUNT;

kfree(filp->private\_data); **//reentrant code 觀念 (本文尚未說明)**

return 0;

};

## Linux 驅動程式觀念解析, #6: 依流程來實作 -- Physical Device Driver

**I/O 存取的觀念**

I/O device須透過I/O port來存取與控制，每個I/O port都會被指定一個memory address，稱為I/O port address(或port address)，此即所謂的memory mapped I/O。memory mapped I/O的意義為，透過I/O port被指定的memory address來存取I/O device，可將複雜的I/O device存取變成簡單的memory存取，也不用assembly存取 I/O device。

Memory-mapped I/O的觀念是將I/O port或I/O memory “mapping” 到 memory address上，此位址稱為I/O port address。採用memory-mapped I/O觀念的主要好處是可以將I/O device的存取變成記憶體存取。因此，對使用者而言，存取I/O裝置就會變成跟CPU的記憶體存取一樣。RISC 架構的處理器在system design方面，也都採取memory-mapped I/O (I/O memory) 的觀念。

**Linux I/O Port 存取介面**

在x86平臺上，I/O port與I/O memory可以看成是一樣的東西。但在學習Linux驅動程式實作時，則要把二者清楚的分開來。若是要存取I/O port，Linux提供以下的I/O port存取介面：

*˙ unsigned inb(unsigned port);  
˙ unsigned inw(unsigned port);  
˙ unsigned inl(unsigned port);  
˙ void outb(unsigned char byte, unsigned port);  
˙ void outw(unsigned short word, unsigned port);  
˙ void outl(unsigned long word, unsigned port);*

若是要存取I/O “memory”，則改用以下函數：

*˙ unsigned readb(unsigned port);  
˙ unsigned readw(unsigned port);  
˙ unsigned readl(unsigned port);  
˙ void writeb(unsigned char byte, unsigned port);  
˙ void writew(unsigned short word, unsigned port);  
˙ void writel(unsigned long word, unsigned port);*

inb()表示要由I/O port address讀取1 byte的資料，outw()表示要輸出1 short word（2 bytes）的資料到指定的I/O port address；同理，readl()表示要由I/O memory address讀取1 long word（4 bytes）的資料，其它函數則依此類推。

範例透過I/O port 80H與debug card溝通，因此只要執行：outb(num, 0x80);

即可將數字”num”顯示在debug card上。

|  |
| --- |
| 在未學習ioremap()函數前，都會以直接存取I/O port的方式來設計。但Linux device driver是「不能直接」存取I/O port或I/O memory的，必須將I/O port或I/O memory “remapping” 到kernel virtual address後才能存取裝置。此觀念在學習 PCI 驅動程式設計時便能看到。 |

**完成我們的範例**

了解 Linux 驅動程式如存取 I/O device 後，我們就可以完成 ops->write 實作了！以下是我們的實作程式碼：

*unsigned long IOPort = 0x80;*

*void write\_card(unsigned int num) {*

*MSG("write 0x%02X (%d) to debug card", (unsigned char)num, num);*

***outb****((unsigned char)num, IOPort);*

*}*

*ssize\_t card\_write(struct file \*filp, const char \*buff, size\_t count, loff\_t \*offp) {*

*char \*str;*

*unsigned int num;*

*int i;*

*if (count == 0) return 0;*

*filp->private\_data = (char \*)kmalloc(64, GFP\_KERNEL);*

*str = filp->private\_data;*

*if (copy\_from\_user(str, buff, count)) return -EFAULT;*

*/\* atoi() \*/*

*num = str[0]-'0'; for (i = 1; i < count; i++) { num = num\*10 + (str[i]-'0'); }*

*write\_card(num);*

*return 1;  
};***完整範例列表**

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

#include <linux/config.h>  
#include <linux/ioport.h>  
#include <linux/errno.h>  
#include <linux/sched.h>  
#include <linux/mm.h>  
#include <asm/io.h>  
#include <asm/uaccess.h>  
#include "card.h"

unsigned long IOPort = 0x80;

int card\_release(struct inode \*, struct file \*);  
int card\_open(struct inode \*, struct file \*);  
int card\_ioctl(struct inode \*, struct file \*, unsigned int, unsigned long);  
ssize\_t card\_write(struct file \*, const char \*, size\_t, loff\_t \*);

void write\_card(unsigned int);

void write\_card(unsigned int num) {

MSG("write 0x%02X (%d) to debug card", (unsigned char)num, num);

outb((unsigned char)num, IOPort);

}

int card\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg) {

switch (cmd) {

case IOCTL\_RESET:

write\_card(0x00);

break;

default:

return -1;

}

return 0;

}

ssize\_t card\_write(struct file \*filp, const char \*buff, size\_t count, loff\_t \*offp) {

char \*str;

unsigned int num;

int i;

if (count == 0) return 0;

filp->private\_data = (char \*)kmalloc(64, GFP\_KERNEL);

str = filp->private\_data;

if (copy\_from\_user(str, buff, count)) return -EFAULT;

/\* atoi() \*/

num = str[0]-'0'; for (i = 1; i < count; i++) { num = num\*10 + (str[i]-'0'); }

write\_card(num);

return 1;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct file\_operations card\_fops = {

open: card\_open,

write: card\_write,

release: card\_release,

ioctl: card\_ioctl,

};

int card\_release(struct inode \*inode, struct file \*filp) {

MOD\_DEC\_USE\_COUNT;

kfree(filp->private\_data);

return 0;

};

int card\_open(struct inode \*inode, struct file \*filp) {

MOD\_INC\_USE\_COUNT;

return 0;

};

int init\_module(void) {

MSG("DEBUG CARD v0.1.1");

if (register\_chrdev(DEV\_MAJOR, DEV\_NAME, &card\_fops) < 0) {

MSG("Couldn't register a device.");

return -1;

}

return 0;

}

void cleanup\_module(void) {

if (unregister\_chrdev(DEV\_MAJOR, DEV\_NAME))

MSG("failed to unregister driver");

else

MSG("driver un-installed\n");

}

MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("[www.jollen.org](http://www.jollen.org/)");

**// card.h**  
#ifndef \_CARD\_H\_

#define MSG(format, arg...) printk(KERN\_INFO "DEBUG CARD: " format "\n", ## arg)

#include <linux/ioctl.h>

#define DEV\_MAJOR 121

#define DEV\_NAME "debug"

#define DEV\_IOCTLID 0xD0

#define IOCTL\_WRITE \_IOW(DEV\_IOCTLID, 10, int)

#define IOCTL\_RESET \_IOW(DEV\_IOCTLID, 0, int)

#endif

**寫 User Program 來測試**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/ioctl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include "card.h"

int main(int argc, char \*argv[]) {

int devfd;

unsigned int num = 0;

if (argc == 1) argv[1] = "0";

devfd = open("/dev/debug", O\_RDWR);

if (devfd == -1) {

printf("Can't open /dev/debug\n");

return -1;

}

printf("Resetting debug card...\n");

ioctl(devfd, IOCTL\_RESET, NULL);

printf("Done. Wait 1 second...\n");

sleep(1);

printf("Writing %s...\n", argv[1]);

write(devfd, argv[1], strlen(argv[1]));

printf("Done.\n");

close(devfd);

return 0;

}

## Linux 驅動程式的 I/O, #1: 基本概念

**Linux 驅動程式 I/O 機制**

Linux device driver 處理 I/O 的「基本款」是：

* fops->ioctl
* fops->read
* fops->write

另外「典藏款」則是 mmap，未來在「Linux Device Driver 進階」專欄裡再來討論這個主題。

**fops->ioctl**

ioctl 代表 input/output control 的意思，故名思義，ioctl system call 是用來控制 I/O 讀寫用的，並且是支援 user application 存取裝置的重要 system call。因此，在Linux驅動程式設計上，我們會實作ioctl system call以提供user application讀寫（input/output）裝置的功能。

依此觀念，回到架構篇所舉的 debug card 範例。當 user application 需要將數字顯示到 debug card 時，範例 debug card 0.1.0 便需要實作 ioctl system call，然後在*fops->ioctl* 裡呼叫 *outb()*將 user application 所指定的數字輸出至 I/O port 80H。

User application 使用 GNU LIBC 的*ioctl()*函數呼叫device driver所提供的命令來「控制」裝置，因此驅動程式必須實作 *fops->ioctl* 以提供「命令」給使用者。

**fops->read & fops->write**

read/write 是 Linux 驅動程式最重要的 2 個 driver function，也是驅動程式最核心的觀念所在。對驅動程式而言，read/write 的目的是在實作並支援 user application 的 read() 與 write() 函數；user application 是否能正常由硬體讀寫資料，完全掌握在驅動程式的 read/write ethod。

User application 呼叫*read()*/*write()*函數後，就會執行 *fops->read* 與 *fops->write*。read/write method 負責讀取使用者資料與進行裝置的I/O 存取。依照觸發資料傳輸的方式來區分，我們可以將 I/O 裝置分成以下 2 種（from hardware view）：

* Polling：I/O裝置不具備中斷。
* Interrupt：I/O裝置以中斷觸發方式進行I/O。

根據I/O處理原理的不同（from software view），可以將 read/write method 的實作策略分成多種排列組合來討論。為了簡化討論內容，未來的日記將鎖定「Interrupt 式的 I/O」來做探討。

## Linux 驅動程式的 I/O, #2: I/O 存取相關函數

**I/O 存取相關函數**

要提到「I/O 處理」當然要整理 Linux 提供的相關函數，以下分 3 大類來整理：

1. I/O port  
2. I/O memory  
3. PCI configuration space  
4. ioremap

**I/O Port**

以下是 Linux 提供最原始的 I/O port 存取函數：

˙ unsigned inb(unsigned port);  
˙ unsigned inw(unsigned port);  
˙ unsigned inl(unsigned port);  
˙ void outb(unsigned char byte, unsigned port);  
˙ void outw(unsigned short word, unsigned port);  
˙ void outl(unsigned long word, unsigned port);

**I/O Memory**

以下是 Linux 提供最原始的 I/O memory 存取函數：

˙ unsigned readb(unsigned port);  
˙ unsigned readw(unsigned port);  
˙ unsigned readl(unsigned port);  
˙ void writeb(unsigned char byte, unsigned port);  
˙ void writew(unsigned short word, unsigned port);  
˙ void writel(unsigned long word, unsigned port);

對於 I/O memory 的操作，Linux 也提供 memory copy 系列函數如下：

˙ memset\_io(address, value, count);  
˙ memcpy\_fromio(dest, source, num);  
˙ memcpy\_toio(dest, source, num);

以上在「[Linux 驅動程式觀念解析, #6: 依流程來實作 -- Physical Device Driver](http://www.jollen.org/blog/2006/05/linux_6_physical_device_driver_1.html)」介紹過一次，並且也搭配了一個簡單範例做說明，您可參考該文。

**PCI Configuration Space**

Linux 也提供讀寫 PCI configuration space（PCI BIOS）的函數：

˙ int pci\_read\_config\_byte(struct pci\_dev \*dev, int where, u8 \*val);  
˙ int pci\_read\_config\_word(struct pci\_dev \*dev, int where, u16 \*val);  
˙ int pci\_read\_config\_dword(struct pci\_dev \*dev, int where, u32 \*val);  
˙ int pci\_write\_config\_byte(struct pci\_dev \*dev, int where, u8 val);  
˙ int pci\_write\_config\_word(struct pci\_dev \*dev, int where, u16 val);  
˙ int pci\_write\_config\_dword(struct pci\_dev \*dev, int where, u32 val);

有些朋友可能看過開頭是 pcibios\_\* 的函數版本，「不過這是舊的函數，請勿再使用」。

**ioremap()**

這個 API 就重要到不行了，任何時候，Linux device driver 都「不能直接存取 physical address」。所以，「使用以上的 I/O 相關函數時，只能傳 virtual address，不能傳入 physical address」，ioremap() 就是用來將 physical address 對應到 virtual address 的 API。

**小結**

對於 I/O 函數的使用，應該在「深諳」Linux驅動程式架構與Linux作業系統原理的情況下使用，「單純的 kernel module + IO APIs」並不叫做 Linux 驅動程式，再更進一步的「kernel module + read/write/ioctl + IO APIs」也只是小聰明（編註），還是稱不上 Linux「驅動程式」。建構在作業系統裡的驅動程式，90% 都是在實作良好的機制與行為，因此「OS 原理與機制的研究」，才是正確的思考方向。與大家分享自己的心得，希望對您的學習有幫助。

編註：這是 Linux device driver 的「開始」但不是全部，也只是冰山一角。但是許多教育訓練機構的課程卻是以此為做為規劃方向，並不是很妥當。

## Linux 驅動程式的 I/O, #3: kernel-space 與 user-space 的「I/O」

**重要觀念**

任何作業系統底下的「驅動程式」，都需要分二個層面來討論所謂的「I/O 處理」：

1. 實體層：驅動程式 v.s. 硬體。

2. 虛擬層：驅動程式 v.s. user process

在前一篇日記「[Linux 驅動程式的 I/O, #2: I/O 存取相關函數](http://www.jollen.org/blog/2006/12/linux_device_driver_io2.html)」中所提到的 I/O 函數是處理「實體層」的 I/O；本日記所要介紹的 *copy\_to\_user()* 與 *copy\_from\_user()* 則是在處理「虛擬層」的 I/O。另外，在繼續往下讀之前，您必須了解以下的觀念都是「等價」的：

1. 驅動程式與 user process 的 I/O；等於

2. 驅動程式與 user process 間的 data communication；等於

3. kernel-space 與 user-space 間的 data communication。

此外，還要了解：

1. user-space 無法「直接」存取 kernel-space 的記憶體。

2. 「Linux device driver」與「user-space」間的 I/O 會與*fops->read*、*fops->write* 與 *fops->ioctl*共三個 system call 有關。

**copy\_to\_user() 與 copy\_from\_user()**

了解以上的觀念後，再來「直接殺進重點」就很容易懂了：從 user-space 讀取資料至 kernel-space，或是將 kernel-space 的資料寫至 user-space，「必須」透過 kernel 提供的 2 個 API 來進行。這二個 API 如下：

˙ long copy\_to\_user(void \*to, const void \*from, long n);  
˙ long copy\_from\_user(void \*to, const void \*from, long n);

參數說明，以 *copy\_to\_user()* 來說：

˙ **to**：資料的目的位址，此參數為一個指向 user-space 記憶體的指標。  
˙**from**：資料的來源位址，此參數為一個指向 kernel-space 記憶體的指標。  
**˙ 口訣：copy data to user-space from kernel-space**

以 *copy\_from\_user()* 來說：

˙ **to**：資料的目的位址，此參數為一個指向 kernel-space 記憶體的指標。  
˙**from**：資料的來源位址，此參數為一個指向 user-space 記憶體的指標。  
**˙ 口訣：copy data from user-space to kernel-space**

由 user-space 讀取資料，或是寫入資料給 user-space 的 3 個 driver method 為：read、write與ioctl。

另外，指向 user-space 的指標是 kernel 回呼 driver method 時所傳遞進來的，可由 read、write 與 ioctl driver function 的函數原型宣告來觀察（紅色部份）：

˙ int card\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long **arg**);  
˙ ssize\_t write(struct file \*filp, const char **\*buff**, size\_t count, loff\_t \*offp);  
˙ ssize\_t read(struct file \*filp, char **\*buff**, size\_t count, loff\_t \*offp);

*fops->ioctl* 的參數 **arg**、fops->write 與 fops->read 的參數 **buff** 是指向 user-space 資料的指標。撰寫程式時，要注意資料型別上的不同。

下一篇日記再寫一個範例來配合著研究，大家應該會更清楚