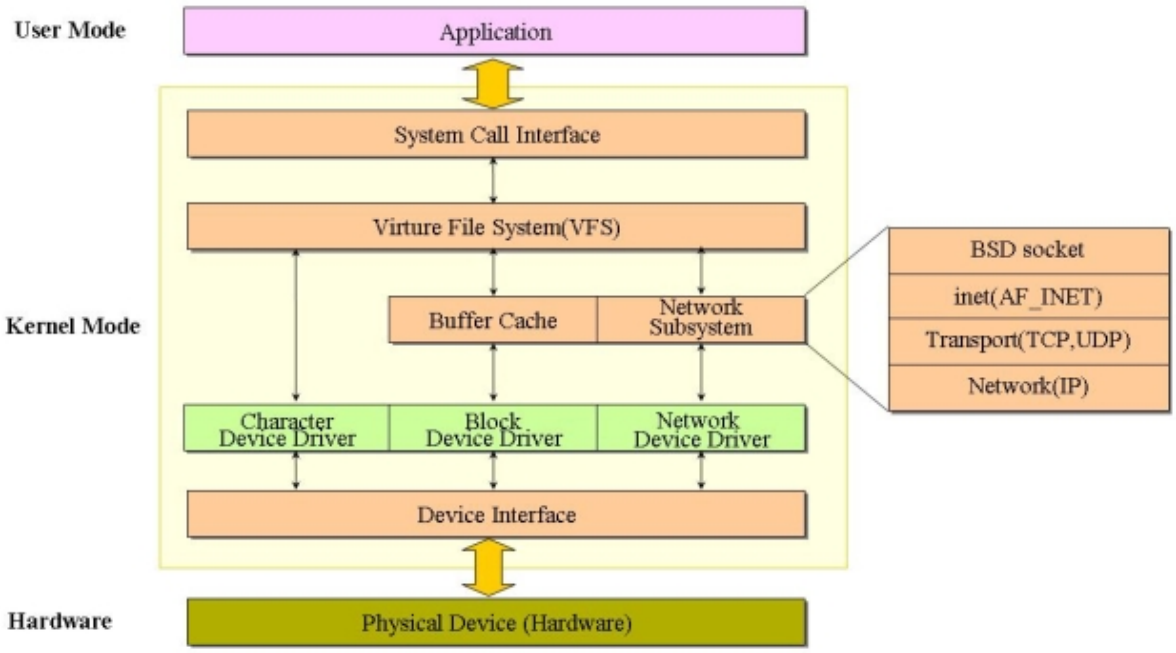
驅動程式的大架構

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。
3. Linux 驅動程式可分為 3 大類型，如下圖綠色部份。



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 的主題重要性大於 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 來設計應用程式是這個主題的重點。與 user application 如何互動，是撰寫驅動程式時所要考慮的重要一環，只考量驅動程式本身的設計，而忽略或輕忽 user application 的設計，是錯誤的觀念。

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。本書秉持這樣的信念撰寫而成，仔細閱讀絕對是多多益善的。

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，本書會以市面上最容易取得的 BT878 視訊擷取晶片為例做說明。

小結

學習 Linux device driver 應由 character device driver 起步，因為許多重要的入門觀念均可藉由 character device driver 學得。

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 的最大優點是使得驅動程式的設計抽象化 (abstraction)，我們可以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。

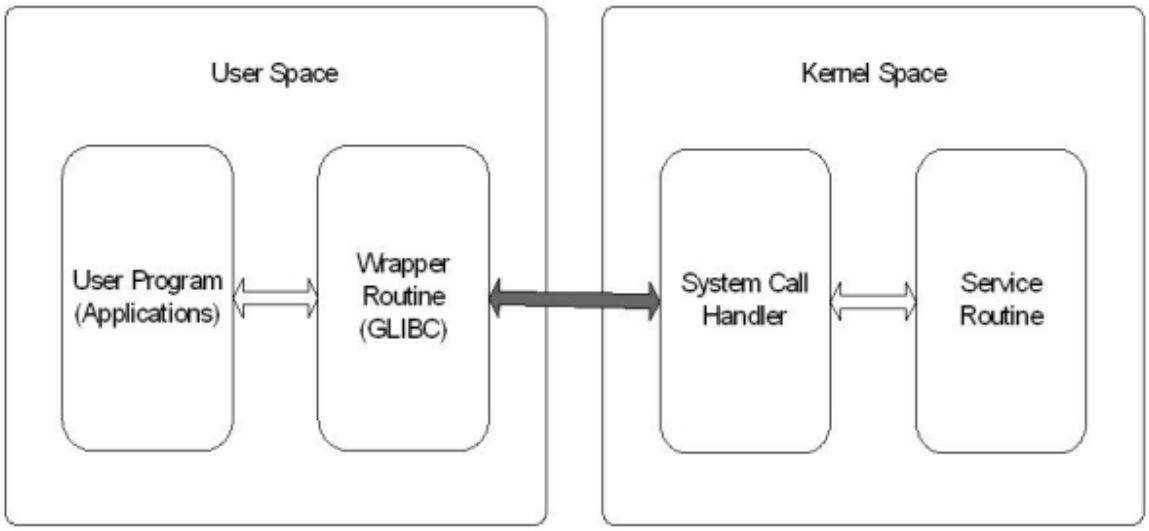
System Calls

Linux 驅動程式的大架構中，system call 是屬於第一層的架構；本文以一個 Linux 範例程式來展示 user application 與 Linux 驅動程式的關係。

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);

   if (devfd == -1) {
      printf("Can't open /dev/debug\n");
      return -1;
   }

   ioctl(devfd, IOCTL_WRITE, num);
   close(devfd);
   return 0;
}
```

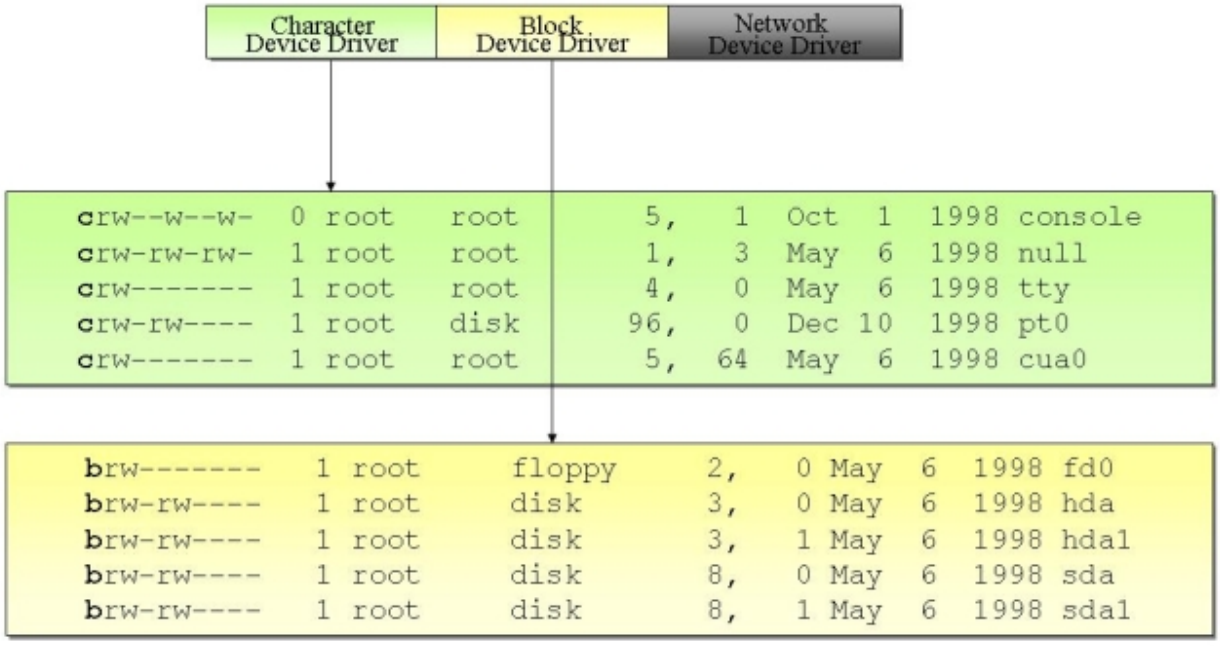
當我們打開/dev/debug檔案時，範例所呼叫open()函數會叫起支援/dev/debug驅動程式的對應函數；同理，我們對/dev/debug執行ioctl()函數時，也會叫起驅動程式的相對應函數。

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。

Device File

什麼是 Device File

Device files 是 UNIX 系統的獨特觀念，在 UNIX 系統底下我們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，或 special files。



Linux device driver 與 user 的重要溝通橋梁為 device files，在 Linux 系統底下，我們看到的 device files 如圖所示。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 代表一個特定的裝置，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

我們在設計 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，我們會指定一個 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 並不會處理不同的 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。

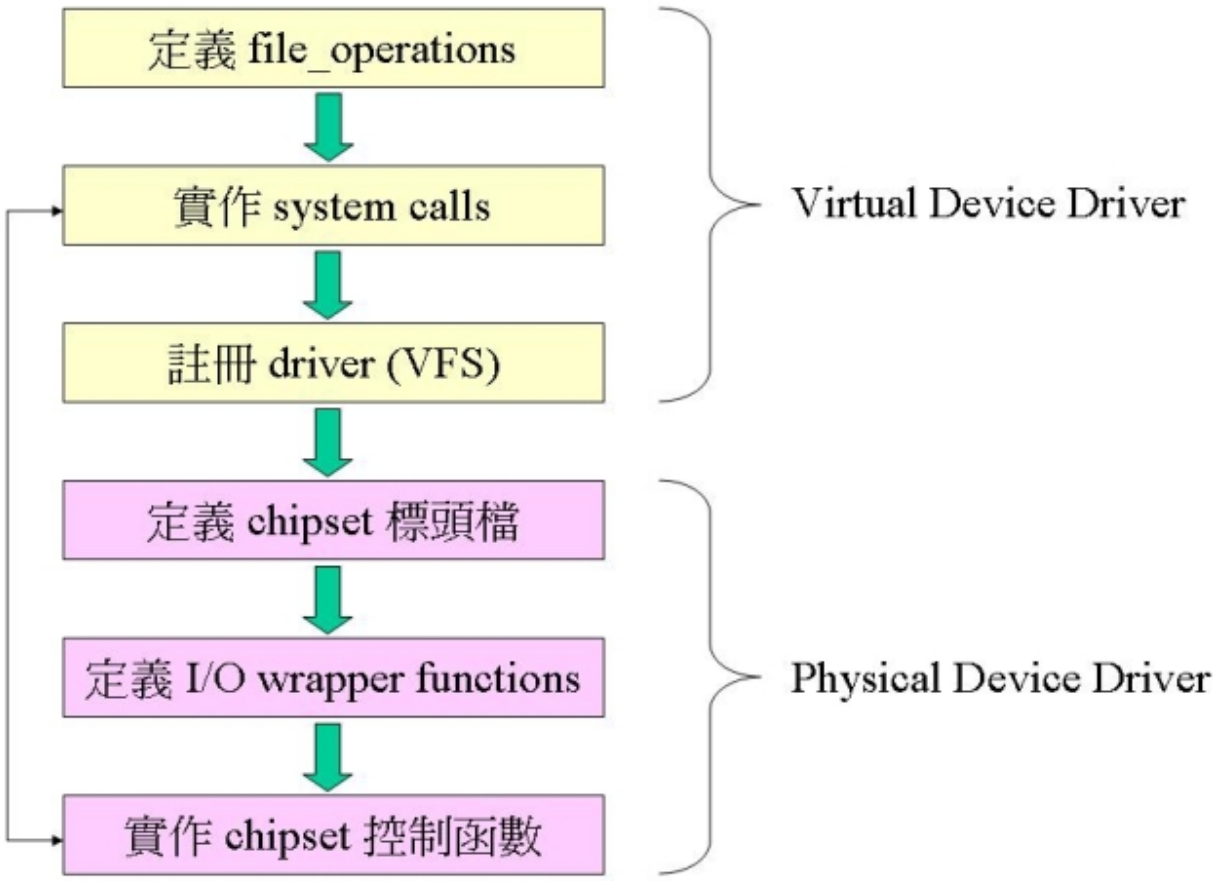
file_operations 是學習 device driver 最重要的一個資料結構，file_operations內的成員為函數指標，指向 “system call 的實作函數”。file_operations 即是圖中的 VFS 層。換句話說，Linux驅動程式是透過file_operations來建構VFS層的支援。而file_operation裡的函數指標，即是指向每一個system call的實作函數。

Linux 驅動程式一般化設計流程

了解重要的架構觀念面後，接著就是 Linux 驅動程式本身了！Linux 驅動程式的設計雖然沒有一定的標準流程，但是由「觀念」層面可以歸納出一個一般化的流程。

一般化設計流程

我們提過，依照驅動程式本身的實作，可以將 Linux 驅動程式分為 2 大部份：virtual device driver與physical device driver。



此流程為一個觀念流程，實際撰寫驅動程式時，並不會完全以這個流程來設計。但初學Linux驅動程式時，則應以此流程為主循序學習，才能理解基本的Linux驅動程式實作。

對Linux驅動程式而言，virtual device driver的重要性遠在physical device driver之上，乍聽之下這或許不太能理解，因為沒有physical device driver是無法真正驅動硬體的。但實作上，physical device driver是一成不變的程式寫法，能不能寫出好的驅動程式，關鍵是在virtual device driver的部份。

struct file_operations

struct file_operations 是 kernel 提供的一個重要資料結構，這是學習 Linux 驅動程式第一個會認識的對象，也是最重要的一個主題。

Linux 驅動程式建構在 file_operations 之上。file_operations定義驅動程式的system call與實作system call的函數，我們把file_operations任何一個部份拿出來討論的話，都能切成virtual device driver與physical device driver二個部份。

流程解說

Virtual device driver往上是為了連結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 的目的在於設計一個「機制」良好的kernel mode驅動程式，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。

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,
};
```

由此定義可以,我們所計的驅動程式將提供 open/write/close(release)/ioctl 4 個 system call 介面給 user application。

實作 System Call

接著要實作我們所提供的4個 system call。open/close(即 release)/read/write/ ioctl 是初學 Linux 驅動程式最重要的 5 個 system call,了解如何實作不同的 system call,是學好 Linux 驅動程式的重要工作。
本文先介紹 open/close(release)/write 的實作。此部份說明如後。

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的動作必須在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,以下是幾個基本的裝置註冊函數:

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");
   MSG("  Copyright (C) 2004 www.jollen.org");

   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

註冊的動作是寫在init_module()裡,因此當使用者執行insmod載入驅動程式時register_chrdev()便會執行。由此可知,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

我們所要設計的 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。

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。

為了方便說明起見,本文後文將以「fops->open」表示實作open system call的driver function。其它system call亦同。

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;
};
```

依流程來實作 -- Physical Device Driver

接續前文的實作,繼續完成 physical device driver 部份;由於 physical device driver 與 I/O 存取密切相關,因此我們會先說明 Linux 的 I/O 存取函數。

I/O 存取的觀念

I/O device必須透過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/Oport address(或port address),此即所謂的memory mapped I/O。

memory mapped I/O的意義為,我們可以透過I/O port被指定的memory address來存取I/O device, 如此可將複雜的I/O device存取變成簡單的memory存取,也不需要使用 assembly 來存取 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,則是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。有些debug card的規格也支援其它的I/O port位址,若要輸出到其它I/O port位址做測試,請自行修改範例。

在未學習ioremap()函數前,我們的範例都會以直接存取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

```
/*
 *  Debug Card 0.1.1 - Port 80 Debug Card Driver
 *
 *  Copyright (C) 2004 www.jollen.org
 *
 *  This file may be redistributed under the terms of the GNU Public
 *  License.
 */


#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");
   MSG("  Copyright (C) 2004 www.jollen.org");

   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");
// 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

```
/*
 *  Debug Card 0.1.1 - Port 80 Debug Card 'User-Space' Driver
 *
 *  Copyright (C) 2004 www.jollen.org
 *
 *  This file may be redistributed under the terms of the GNU Public
 *  License.
 */


#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;
}
```

觀念大考驗

到這裡為止，我們已經完成階段性任務了--了解 Linux 驅動程式的架構觀念。

下一篇文章，我們會具體描繪出此範例的執行流程路徑；透過這張圖，大家便能考驗自己是否已經了解主要的驅動程式架構觀念了！

觀念大追擊

您是否能看圖說明範例的觀念。

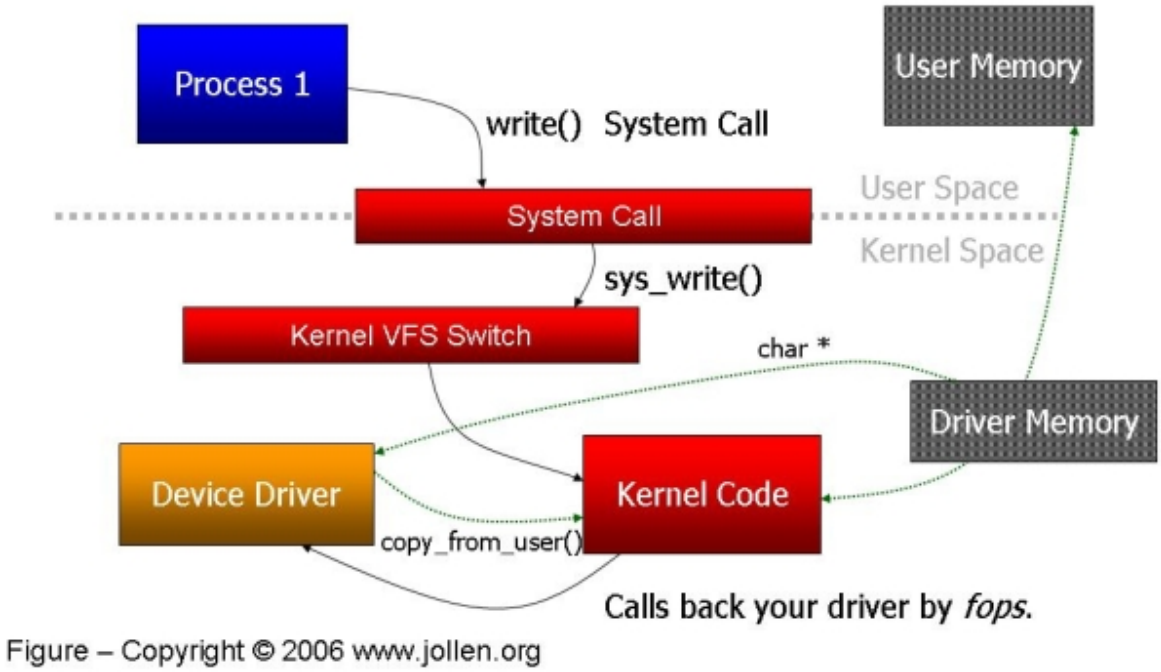


Figure – Copyright © 2006 www.jollen.org

TIP
fops所指的driver function其實是被Linux kernel所「回呼」（callback）。

Linux驅動程式將fops「註冊」至kernel裡後，並不是被user application直接呼叫，而是透過system call interface，因此fops所指的函數應是被kernel回呼。

Callback的機制有一個好處是，當函數被呼叫時，表示此時系統符合該函數被回呼的條件。因此，driver function可以預期自己是在符合一些條件的環境下執行。Callback機制另一個特點是，Linux kernel會傳遞「適當」的參數給driver function，driver function可以直接使用所接收的參數資料。

這張圖是範例 (debug card 0.1.1) 的執行圖 (Execute Flow/Path)，如果您能根據範例程式清楚地說明此圖，表示您已經掌握最主要的 Linux 驅動程式觀念了！

Linux 驅動程式的 I/O, #1: 基本概念

由本篇日記開始，我們將進行「Linux Device Driver 入門：I/O 處理」的議題討論。這裡所提的 I/O 處理定義是：user process 與 physical device 的 I/O 存取。

在讀「Linux Device Driver 入門：I/O 處理」專欄前，您必須熟悉 Linux 驅動程式的架構，因此「Linux Device Driver 入門：架構層」的專欄是 Jollen's Linux Device Driver 系列專欄的先備知識；此外，接下來的專欄使用的語法也必須對架構層有基本認識後才能看得懂。

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」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」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。