## <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分成3種類型：character device driver, block device driver, network device driver。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，善用Linux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。

**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)，不需要太深入硬體層次。

Linux kernel所提供的API經過良好設計，使用 kernel所提供的API可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間透過system call溝通，實際上user application以device file與裝置驅動程式溝通。驅動程式須建構在此「file」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 = open("/dev/debug", O\_RDONLY);  
 ioctl(devfd, IOCTL\_WRITE, num);  
 close(devfd);  
 return 0;  
}*

範例中的open()與ioctl()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，需由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**

device driver會先透過“註冊” (register)的動作將自己註冊到kernel裡，註冊時會指定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是device driver內部使用，kernel不會處理不同的minor number。

設計device driver的第一步要定義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VFS層。換句話說，Linux驅動程式透過file\_operations支援VFS層。而file\_operation裡的函數指標，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**

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實作的system call進入點。實作system call的函數便透過file\_operations結構來定義稱實作system call的函數為driver method。

kernel會在需要時回呼(callback)註冊的driver method。當driver裡的method被呼叫時，kernel傳遞參數給driver method，driver method可由kernel傳遞進來的參數取得驅動程式資訊。註冊driver的動作藉由register\_chrdev()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」的動作觀念上是將fops加到kernel的VFS層，因此user application須透過「device file」呼叫driver method。註冊的另一層涵意是將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。理論上可將晶片的暫存器分成3大類：1. data registers。2. control registers。3. status registers。Data register是晶片存放資料的暫存器，control register是控制晶片的暫存器，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，需了解硬體使用的晶片組，晶片組需參考IC設計廠提供的「datasheet」來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)。

驅動程式要透過control register來控制晶片，需隨時查閱晶片的datasheet，了解每個暫存器的用途。通常暫存器的每個位元(bit)都有特定用途，因此設計驅動程式時，須熟悉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，通常可從vendor取得。接著再定義一組操作暫存器的I/O函數，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，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，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 MSG(format, arg...) printk(KERN\_INFO "DEBUG CARD: " format "\n", ## arg)*

*int init\_module(void){*

*if (register\_chrdev(121,”debug”, &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：為讓device在/dev file system下的名稱。  
˙ 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標頭檔及I/O wrapper function**

設計的Port 80H除錯卡驅動程式，不需定義標頭檔及I/O wrapper function；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。

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。System call的實作即driver function要負責處理的基本工作。

**open System Call**

fops->open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，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，就可完成 ops->write 實作！以下是實作程式碼：

*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 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;*

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

*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) {*

*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：1. fops->ioctl。2. fops->read。3. fops->write。

**fops->ioctl**

ioctl代表input/output control，故名思義，ioctl system call是來控制I/O的讀寫，並支援user application存取裝置的system call。User application使用GNU LIBC的*ioctl()*函數呼叫device driver提供的命令來「控制」裝置，因此驅動程式須實作 *fops->ioctl* 以提供「命令」給使用者。

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

驅動程式read/write的目的是實作並支援user application的read()與write()函數；user application能否正常由硬體讀寫資料，完全掌握在驅動程式的 read/write method。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，將I/O裝置分成2 種(from hardware view)：1. Polling：I/O裝置不具備中斷。2. Interrupt：I/O裝置以中斷觸發方式進行I/O。

## 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」，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. 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);*

**ioremap()**

這個API很重要，任何時候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physical address」，ioremap() 是用來將physical address對應到virtual address的API。

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

**重要觀念**

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。前一篇中提到的I/O函數是處理「實體層」的I/O；*copy\_to\_user()* 與 *copy\_from\_user()*則是處理「虛擬層」的 I/O。驅動程式與 user process 的 I/O；等於驅動程式與user process 間的data communication；等於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 driver」與「user-space」間的I/O與*fops->read*、*fops->write* 與 *fops->ioctl*三個system call有關。

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

從user-space讀取資料至kernel-space，或是將kernel-space的資料寫至user-space，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。

**I2C**

The I2C communication can’t be done by just calling *read()* and *write()*. To read/write data due to I2C is calling *ioctl()* to access the device file of */dev/i2c-0, /dev/i2c-1*,… Each registered i2c adapter gets a number, counting from 0. Examine */sys/class/i2c-dev/* to see the corresponding between the number and adapter. I2C device files are character device files with major device number 89 and a minor device number should be called "*i2c-%d*" (*i2c-0, i2c-1, …, i2c-255*).

The implementation of SMBus is mostly based on the I2C specification and not required to add extra pins. Compared to I2C bus, SMBus adds some new functions and the bus timing is different from the one of I2C.

There are two files named "*i2c-dev.h*", one is distributed with the Linux kernel and is meant to be included from kernel driver code, to open the device file, as follows: *int file = open("/dev/i2c-2", O\_RDWR);*

When you have opened the device, you must specify with what device address you want to communicate:

*int addr = 0x40; /\* The I2C address \*/*

*if (ioctl(file, I2C\_SLAVE, addr) < 0) { /\* ERROR HANDLING; you can check errno to see what went wrong \*/exit(1); }*

Use SMBus commands, which are preferred if the device supports them, or plain I2C to communicate with your device.

*\_\_u8 register = 0x10; /\* Device register to access \*/*

*\_\_s32 res = i2c\_smbus\_read\_word\_data(file, register); /\* Using SMBus commands \*/*

*if (res < 0) {/\* ERROR HANDLING: i2c transaction failed \*/} else {/\* res contains the read word \*/}*

*/\* Using I2C Write, equivalent of i2c\_smbus\_write\_word\_data(file, register, 0x6543) \*/*

PCA9555 is a 16 GPIO extender on I2C bus. The Linux GPIO lib is easy to extend: 1. Wire the chip on the I2C bus, 2. Add the configuration in your code. Add the device to your I2C bus declaration like:

*static struct i2c\_board\_info \_\_initdata bfin\_i2c\_board\_info[] = {{ ... }, { ... }};*  
Just add to the array your chip: *I2C\_BOARD\_INFO("pca9555",0x20)*…

0x20 is the I2C address of PCA9555. It’s configured by the pin A0, A1, A2, with all three are connected to zero volt. Check the PCA support in the kernel config menu (device->gpio), compile, run it and done! The new GPIO is added at high number (240 in my case). New GPIO extender should be in */sys/class/gpio/*.

**Platform device and driver**:

In kernel mode, Linux provides a set of functions to create/destroy/handle the device file in *sysfs*. Use *class\_create()* to create a class under */sys/class*:

*struct class\* my\_class = class\_create(THIS\_MODULE, “my\_class”);* -> */sys/class/my\_class*

Exploit *device\_create()* to create a device under the class:

*struct device\* my\_device v= device\_create(my\_class, NULL, 0, NULL, “my\_device%d”, 0);* -> */sys/class/my\_class/my\_device*. (<http://hi.baidu.com/sdqdshixin/item/bb833382c7521aeae596e0c3> for more detailed info about *device\_create*()). Mind that there are also some other files like *uevent*, *subsystem* created concurrently. Mind that the third argument of *device\_create()* is the device number, and you can also send 0 in certain conditions, but the second argument of device\_destroy() must be 0 too. Set the structure as the forth parameter of *device\_create()*, so that other functions can also read this structure through *dev\_get\_drvdata()*. *xxx\_show()/xxx\_store()* get the structure in this way. To set/get *device* and *platform\_device* data easily, the data structure is defined like:

*struct BIOS\_INFO\_DATA{*

*char\* name; // Keep track of the device name from platform\_device::dev::name*

*struct device \*dev;*

*char bios\_name[32];*

*};*

*struct BIOS\_INFO\_PLATFORM\_DATA{*

*struct BIOS\_INFO\_DATA \*pdata;*

*struct platform\_device \* pdev;*

*};*

Finally call the *device\_create\_file()* to create device file under the device:

device\_create\_file(*my\_device, &dev\_attr\_devicefile1); -> /sys/class/my\_class/my\_device/devicefile1*

Mind *dev\_attr\_devicefile1* should be defined beforehand:

*char devicefile1\_str[] = “devicefile1”;*

*static DEVICE\_ATTR(devicefile1, 0777, devicefile1\_show, NULL);*

where *devicefile1\_show()* is the function that defines the content which is shown in user mode.

size\_t *devicefile1\_show(struct device\* dev, struct device\_attr\* attr, char \*buf){*

*memcpy(buf, devicefile1\_str, sizeof(devicefile1\_str) / sizeof(devicefile1\_str[0]));*

*return sizeof(devicefile1\_str) / sizeof(devicefile1\_str[0]);*

*}*

在内核中sysfs 属性一般是由 *\_\_ATTR* 系列的宏来声明的，如对设备的使用 *DEVICE\_ATTR*，对总线使用 *BUS\_ATTR*，对驱动使用 *DRIVER\_ATTR*，对类别使用 *CLASS\_ATTR*, 这四个宏来自于*<linux/device.h>*：

*static ssize\_t store\_scan(struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t count){…};*

*static DEVICE\_ATTR(scan, S\_IWUSR, NULL, store\_scan);*

*DEVICE\_ATTR*宏声明有四个参数，分别是名称、权限位、读函数、写函数。这里对应的名称是*scan*，权限是只有属主可写(*S\_IWUSR*)、没有读函数、只有写函数。因此读写功能与权限位是对应的。*scan*属性写入功能是在*store\_scan*函数中实现的，这个接口的四个参数中， buf/count 代表用户写入过来的字符串。

*#define DRIVER\_ATTR(\_name, \_mode, \_show, \_store)*

*struct driver\_attribute driver\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store)*

By using *platform\_device\_register()*, the device node is created in path */sys/device/platform*. For example:

*static struct platform\_device bios\_info\_device = {*

*.name = “bios\_info”,*

*.id = -1,*

*.dev = {*

*.platform\_name = NULL,*

*.release = bios\_info\_device\_release, // This function will be called while invoking platform\_device\_unregister().*

*},*

*};*

The device file */sys/device/platform/bios\_info*. There are also lots of device files created automatically under *bios\_info*. Exploit *platform\_driver\_register()* to register the platform driver like:

*static struct platform\_register bios\_info\_register = {*

*.probe = bios\_info\_driver\_probe,*

*.remove = bios\_info\_driver\_remove,*

*.suspend = bios\_info\_driver\_suspend,*

*.resume = bios\_info\_driver\_resume,*

*.driver = {*

*.name = “bios\_info”,*

*.owner = THIS\_MODULE,*

*},*

*};*

If *platform\_device::name* and *platform\_register::driver::name* are identical, the *bios\_info\_driver\_probe* will be invoked so that you can create the device file in this function.

In implementing character device, register/unregister the driver in *xxx\_init()/xxx\_exit()* by calling *register\_chrdev\_region()/release\_chrdev\_region()* and create/destroy device file in *xxx\_init()/xxx\_exit()* too. In implementing platform device, the creating/destroying device files are implemented in *xxx\_probe()*.

*platform\_set\_drvdata()* sets the data to private data: *pdev->dev.driver\_data = bios\_info\_data;*

*xxx\_probe(platform\_device\* pdev){*

*struct bios\_info\_data\* data = new bios\_info\_data();*

*platform\_set\_drvdata(pdev, bios\_info\_data);*

*}*

Use *dev\_name* to get the value of *platform\_device::name*.

To create a file in user mode, use *open()* so that the kernel creates a system call which invokes *sys\_open()* in kernel mode. To create a file in kernel mode, use *filp\_open()* instead. 在Linux kernel中读写文件没有标准库可用，需利用kernel的函数：*filp\_open(), filp\_close(), vfs\_read() vfs\_write()，set\_fs()，get\_fs()*等，这些函数在*linux/fs.h*和*asm/uaccess.h*头文件中声明。

1. 打开文件: *filp\_open()*在kernel中可以打开文件，其原形如下：

*strcut file\* filp\_open(const char\* filename, int open\_mode, int mode);*

函数返回*strcut file\**结构指针，返回值*IS＿ERR()*来检验其有效性。*filename*：表明要打开或创建文件名称(包括路径)。*open\_mode*：文件的打开方式，与标准库中的*open*相应参数类似:*O\_CREAT,O\_RDWR,O\_RDONLY*等。mode：创建文件时使用，设置创建文件的读写权限，其它情况可以匆略设为0。

2. 读写文件: kernel中文件的读写可使用*vfs\_read()*和*vfs\_write()*。

*ssize\_t vfs\_read(struct file\* filp, char \_\_user\* buffer, size\_t len, loff\_t\* pos);*

*ssize\_t vfs\_write(struct file\* filp, const char \_\_user\* buffer, size\_t len, loff\_t\* pos);*

注意第二个参数*buffer*，前面都有*\_\_user*修饰符，要求这buffer指针都应该指向用空的内存，若对该参数传递kernel空间的指针，函数会返回失败*-EFAULT*。在Kernel中不容易生成用户空间的指针。要使这两个读写函数使用kernel空间的buffer指针也能正确工作，需要使用*set\_fs()*函数：*void set\_fs(mm\_segment\_t fs);*

该函数改变kernel对内存地址检查的处理方式，参数*fs*只有两个取值：*USER＿DS*，*KERNEL＿DS*，分别代表用户和内核空间，kernel默认值为*USER\_DS*，即对用户空间地址检查并做变换。要对内存地址做检查变换的函数中使用内核空间地址，需使用*set\_fs(KERNEL\_DS)*进行设置。*get\_fs()*是取得当前的设置，这两个函数的一般用法为：

*mm\_segment\_t old\_fs;*

*old\_fs = get\_fs();*

*set\_fs(KERNEL\_DS);*

*...... //与内存有关的操作*

*set\_fs(old\_fs);*

其它的内核函数也有用*\_\_user*修饰的参数，在kernel中需要用kernel空间的内存代替时，都可以使用类似办法。使用*vfs\_read()*和*vfs\_write()*最后需要注意的一点是最后的参数*loff\_t\* pos*，pos所指向的值要初始化，表明从文件的什么地方开始读写。

3. 关闭读写文件

*int filp\_close(struct file\*filp, fl\_owner\_t id);*

第二个参数一般传递NULL值，也有用current->files作为实参的。

In Linux kernel, string-accessing functions are also defined, like *strlen()*,… Check *linux/include/string.*h for more detail.