4.1 撰寫 Module 程式

4.1.1 動機與目的：

準備撰寫裝置驅動程式。因為裝置驅動程式是屬於系統核心(Kernel)的一部份，且需要能夠於系統核心處於執行的狀態下，加入系統核心或由系統核心中被移除。故需要被撰寫成模組（Module）型態的程式碼，並且被編譯成模組程式。

4.1.2 如何撰寫模組程式原始碼：

Module 程式的原始程式碼具有特定的型態。如同圖一所示：一個 Module 程式必須要具備有兩個程式：模組啟動與移除兩個程式。這兩個程式具有特定的函式名稱。模組啟動程式的函式名稱必需是 init_Module；而模組移除程式的函式名稱

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h> /* For printk(.) */

int init_module(void) /* 模組啓動 */
{
  printk("<1>Execute init_module(.)\n"); /* 輸出訊息於 Consol 上 */
  return 0;
}

void cleanup_module(void) /* 模組移除 */
{
  printk("<1>Execute cleanup_module(.)\n"); /* 輸出訊息於 Consol 上 */
  return;
}
```

圖 4-1Module 程式範例

必須是 cleanup_module。

不過，於系統核心 2.4 版之後，具有兩個新加入的巨集 module_init 與 module_exit。這兩個巨集讓程式設計師可以使用自訂的函式名稱，作為模組啟動函式與模組移除函式的名稱。此兩個模組均放置於標頭檔：linux/init.h 中。如圖二所示的範例程式，此程式與圖一所示的程式功能相同，但是使用 module_init 與 module_exit 這兩個巨集。因此，範例中的模組啟動程式與模組移除程式，就

可以由程式設計師自行命名。不過請注意，函式的參數部分還是必須依據固定的方式寫作，也就是均不得具有參數。
此兩個範例程式都於程式主體中呼叫 printk 函式，此函式的功用與用法都如同

```
#define MODULE
#include <linux/module.h>
#include <linux/init.h>   /* for module_init and module_exit

int my_init(void) /* 模組啓動 */
{
  printk("<1>Execute init_module(.)\n"); /* 輸出訊息於 Consol 上 */
  return 0;
}

void my_cleanup(void) /* 模組移除 */
{
  printk("<1>Execute cleanup_module(.)\n"); /* 輸出訊息於 Consol 上 */
  return;
}

module_init( my_init ); /* 宣告 my_init 爲此模組之 init_module 函式 */
module_exit( my_cleanup );  /* 宣告 my_cleanup 爲此模組之 */
                            /* cleanup_module 函式          */
```

圖 4-2 使用 module_init 與 module_exit 兩巨集的模組程式範例

printf 函式，但是使用上主要的不同點，就是此函數無法處理浮點數以及此函數是供給系統核心呼叫使用。

4.1.3 編譯模組程式原始碼

編譯模組程式程式原始碼時，必須要使用參數 c，同時也必須使用與愈載入執行此模組的系統核心相同版本的 include 檔案。例如欲將此模組編譯後於 2.4 版的系統核心上執行，則編譯時所使用的引入檔案必須是屬於 2.4 版所附的 include 目錄中的標頭檔。圖三為一個編譯的範例，此範例使用 gcc 這個編譯程式將圖一所示的範例程式（令其儲存於 demo1.c）加以編譯成 object 檔案，成為一個模組程式。其中 gcc 還使用其他的參數，包括 D、I、O、W。D 參數定義一個“__KERNEL__”符號。此符號是供編譯器選擇要編譯標頭檔中與核心有關的程式碼；I 參數是告知編譯器正確的標頭檔所在之路徑；O 參數是告知編譯器要對程

```
Host53>gcc -D__KERNEL__ -I/usr/src/linux/include -O2 -Wall
 -c demo1.c -o demo1.o
Host53>
```
圖 4-3 編譯模組原始碼的範例

式中的 online code 於編譯時就確實展開，其後所跟隨的數字是代表等級，數字越高等級越高，建議不要超過 2；Wall 參數則是要編譯器嚴格檢查程式碼。

4.1.4 模組載入

當一個模組程式已經被編譯成功，成為一個 object 檔案後，就可以將此 object 檔案載入系統核心中。將模組載入系統核心的方法很簡單，只要使用 insmod 這個指令就可以了，不過要注意的是：此指令只有具有 root 權限的使用者可以使用，因此必須先改為以 root 簽入才行。整個指令的使用方式如圖四所示，由此圖中可以看到使用者已經具備了 root 權限。載入時，insmod 的參數就是欲載入模組的 object 檔案的名稱，在此例中為 demo1.o。當載入成功後於 consol 上就可見到模組程式所輸出的訊息：Execute init_module(.)。另外此訊息也會被輸出寫入到/var/log/messages 檔案中。因此若是在 xwin 的環境中，或者不是在 consol 上執行，因為無法看到輸出的訊息，則可以藉由 cat 出該檔案的內容，就可以觀看到該模組已輸出的訊息。

```
Host53>insmod demo1.o
Host53>Execute init_module(.)
Host53>
```
圖 4-4 載入模組的範例

當模組載入後，可以用 lsmod 指令列出所有已經載入的模組名稱，此時應該可以看到名為 demo1 的模組名稱。

4.1.5 模組移除

模組的移除過程，基本上雷同於模組的載入過程。但使用的命令為 rmmod。圖五就是一個移除已經載入模組的範例。此例中移除的模組其目的檔為 demo1.o，被載入後就以 demo1 為其模組名稱，所以移除時，rmmod 的參數就是欲被移除

```
Host53>rmmod demo1
Host53>Execute cleanup_module(.)
Host53>
```
圖五、移除模組的範例

的模組名稱：demo1。請特別注意，不是用 demo1.o 喲。

4.2 基本的驅動程式架構

如前所述，一個驅動程式必須被撰寫成模組的型態，才能夠具備供使用者載入與移除的特性。因此一個驅動程式必然會具有載入模組函式與移除模組這兩個函式。故此兩個函式就分別載入驅動程式與移除驅動程式的角色。

一個驅動程式的存在，主要就是要提供使用某一特定週邊裝置的功能，包括了讀取與設定該週邊的狀態，傳送資料給該周邊，以及由該週邊讀取資料。這些功能都必須透過程式碼來實際達成。程式設計師透過硬體的使用手冊可以知道如何設定、讀取硬體的狀態與資料。因此可以實做出有用的程式碼來運用這些功能。這些程式碼對應於不同的硬體功能，就成為一個個獨立的函式。載入模組函式的功能就在於透過一個特定的制式方式，將這些功能函式提供給系統核心來呼叫使用，並且建立一個可以讓應用程式使用這些功能的管道。

這個制式方式，是運用系統核心所提供的特定函式，將一個內容為指向各功能函式的指標的表格結構，置入系統核心中。此特定函式通常名為 register_xxxx，其中 xxxx 視為特定的文字，例如在附錄一中，因為要製作的周邊裝置驅動程式是屬於字元式裝置，故 xxxx 的部分就是 chrdev。另外，此表格結構為核心已經定義過的一個結構，名稱為 **struct file_operations**。此結構被描述在 linux/fs.h 標頭檔。這個結構內有許多的欄位（fields），如圖六所示，此圖中與撰寫驅動程式有關的欄位已經被標示為粗體字型，包括了***owner**，***read**，***write**，***open**，***release**，***ioclt**。程式設計師透過 open 函式，提供應用程式初始化週邊的能力；透過 release 函式提供應用程式宣告停止使用週邊的能力；透過 ioctl 對週邊裝置下命令；透過 read 與 write 函式對週邊裝置讀取與寫入資料。

再附錄一中的 init_module 函式，就是一個最簡單的周邊裝置模組驅動程式的模組載入函式，在此函式中，‘SET_MODULE_OWNER(&my_fops);’這行巨集指令，其目的就是為了設定 my_fops 這個 struct file_operations instance 的 owner 欄位，而這也是一個標準的設定方式。這個巨集被描述於 linux/fs.h 中。為了使 struct file_operations instance 不會因為 init_module 程式執行完後就壽終正寢，一般可接受的做法中，將此 struct file_operations instance 宣告成 global 型態。而相關欄位的填寫就如同附錄一的程式所示，參考該程式，例如欲將實際撰寫的 open 函式指標填入 open 變數，只要簡單的填入：‘**open: my_open**’，第一個 open 指示要填入 open 欄位，接著是冒號‘：’，然後是實際執行該功能的函式名稱，在此為 SD_CJY_open。

當執行 register_chrdev 函式來對系統註冊這些供某特定週邊使用的函式時，除了要將 struct file_operations instance 的指標當作參數，還需要給予一個名稱以及一個數字來代表此組 instance，也就是代表此該類型裝置的驅動程式。此數字稱為 major number 必須是介於 0~255（2.0 版及之前版本系統核心則介於 0~127），但是頭尾兩個數字 0,255 或 127 是被保留。使用時不可與目前已經使用的數字相衝突。為了避免採用到已經使用的 major number。可以使用自動配置的方式獲取 major number。方法很簡單，只要於註冊時將 register 函數傳送 major number 的參數以零傳送，register 函數就會自動選用一個 major number 並且將選用的值透過函式回傳值將其回傳給呼叫的函式。附錄一的程式就是採用此種方法獲取 major number，完成登錄的動作。

在 Linux 系統中，一般是透過一個稱為 Device number 的數字來表示一個實際的

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);    int (*flush) (struct file *);      int
(*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned
long, unsigned long);
};
```

圖 4-6struct file operations

周邊裝置。此 Device number 是由兩個數字所組成：Major Number 與 Minor Number。Major Number 的部分已經在前面敘述過，Minor number 是透過節點檔案(Node file)所建立。當一個驅動程式註冊成功之後，一個 Major nuber 已經獲

得，但是此時雖然已經讓系統核心知道如何去獲取裝置的資源，但是應用程式還不能夠與此裝置相溝通。也就是應用程式還無法使用此裝置。為了使應用程式能夠使用此裝置，必須要建立一個特殊的檔案。應用程式透過開啟此特殊的檔案，就能夠與此檔案對應的周邊裝置互動。要建立一個 Node File，首先必須先將對應的驅動程式載入，也就是透過 insmod 指令將編譯好的驅動程式模組掛載入系統核心中。此時可以由/proc/devices 檔案中獲知載入模組的 Major number。接著就可以透過 mknod 指令建立 Node File。此 mknod 命令有四個參數，第一個參數為欲建立的 Node File 檔案名稱；第二個參數為檔案的性質，對於字元式裝置驅動程式，使用 'c'；第三個參數為 major number，此必須使用註冊時所用或所得的 major number，也就是剛才在/proc/devices 檔案中所觀察到的數字；第四個參數就是 minor number，此數字介於 0~255 之間，由建立 Node file 時指定，指定時要注意，使用相同 major number 的 Node file 其 minor number 不得相同。因為一個 Node file 就代表一個實體裝置。例如有個裝置被新加入系統中，註冊時獲得 Major number 為 123，則建立 Node file 時，使用 mknod 指令的方式就可以如下：

```
root#mknod /dev/mydevice01 c 123 1
root#
```

執行此命令之後，一個名為 mydevice01 的 Node file 就被建立在/dev 目錄中，其代表使用以 major number 123 註冊的驅動程式。當應用程式使用 open 敘述開啟此檔案時，就可以與此驅動程式建立關聯。

一個簡單但完整的驅動程式原始碼範例被列出於附錄一中。編輯的程序已經被撰寫成一個 makefile，並將之列出於附錄二中。附錄三列出一個 script file 執行檔，該檔案將會完成數項工作：載入模組、找出 major number、建立 node file。附錄四也是一個 script file 執行檔，但是此執行檔將會移除載入的模組、消除對應的 node file。附錄五則是一個使用驅動程式的應用程式範例。該應用程式將會開啟一個節點檔案（Node File），傳送資料給驅動程式，透過驅動程式獲取資料，傳送命令給驅動程式，以及結束驅動程式的使用。這些範例可以用作撰寫驅動程式與使用驅動程式的範本。只要程式設計師加入自己的程式碼於各函式主體中就可以了。

4.3 應用程式與驅動程式間的資料傳輸

應用程式的執行空間並不同於系統核心程式的執行空間。應用程式的執行空間是位於 user space 而系統核心程式的執行空間是位於 kernel space。因此要將 user space 中某一區域的資料複製到 kernel space 中，獲反向而行，需要透過特定函式的幫助才能夠達成。能夠將資料由 user space 中某一區域的資料複製到 kernel space 中，要透過 copy_from_user 函式。反之，要將 kernel space 中某一區域的資料複製到 user space 中，則要透過 copy_to_user 函式的幫助。參看附錄一的程式，

在這個程式中，有一個 SD_CJY_write 函式，於此函式中存在有一段敘述：

```
if(copy_from_user( buf_in_kernel, buf, length ))
{
   return -EFAULT; /* Error */
}
length_write=length;
```

在 if 敘述的條件描訴部分就是 copy_from_user 的一個使用例子。由此例中可以看出此函式需要三個參數，第一個參數為一個指標，其指向資料欲放置區域的第一個位址，此區域是位於 kernel space；第二個參數也為一個指標，其指向資料欲讀取區域的第一個位址，此區域是則是位於 user space；第三個參數則是代表欲複製的資料區域的長度，以位元組（Byte）為單位。而此函數的回傳值則代表尚未處理的位元組數目。另外，於 SD_CJY_read 函式，則存在有一段敘述：

```
if(copy_to_user(buf,data_in_kernel,length)){
   return -EFAULT; /* Error */
}
length_read=(ssize_t)length;
```

在 if 敘述的條件描訴部分就是 copy_to_user 的一個使用例子。由此例中可以看出此函式需要三個參數，第一個參數為一個指標，其指向資料欲放置區域的第一個位址，此區域是則是位於 user space；第二個參數也為一個指標，其指向資料欲讀取區域的第一個位址，此區域是位於 kernel space；第三個參數則是代表欲複製的資料區域的長度，以位元組（Byte）為單位。而此函數的回傳值則代表尚未處理的位元組數目。

4.4 結論

欲撰寫一個裝置驅動程式於 Linux 系統中，首先必須要以撰寫模組程式的格式來撰寫。而裝置的各個功能，必須將其區分為控制，資料傳輸，初始和收尾四個部分。初始部分的功能要撰寫放置於 open 函式；收尾部分的功能要撰寫放置於 release 函式；控制功能要實做於 ioctl 函式；資料傳輸部分的程式碼則要編寫於 read 與 write 函式中。透過附錄一所展示的驅動程式範例，希望能夠給予讀者一個更清楚的輪廓，藉由附錄二到五的程式，希望展現給讀者一個完整的編輯與使用流程。若讀者對此部分已經了解，便可以往更深入的題材研讀，例如撰寫 PCI 介面裝置的驅動程式，網路設備的驅動程式，硬碟裝置的驅動程式等等。希望這個簡單的入門教材，能夠給予讀者於撰寫裝置驅動程式上提供的實質的幫助。

實驗:

範例程式：

SimpleDeviceModule.c:

```
#define __KERNEL__
```

```
#define MODULE

#include<linux/config.h>
#ifdef CONFIG_SMP
#define __SMP__
#endif

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <asm/uaccess.h>

/* my function declaration */
int SD_CJY_open( struct inode *, struct file* );
int SD_CJY_close( struct inode *, struct file* );
ssize_t SD_CJY_read( struct file*,    char*, size_t, loff_t*);
ssize_t SD_CJY_write( struct file*, const char*, size_t, loff_t*);
int SD_CJY_ioctl( struct inode*, struct file*, unsigned int, unsigned long );

/* Instance of file operations struct */
struct file_operations my_fops={
  open: SD_CJY_open,
  release: SD_CJY_close,
  read: SD_CJY_read,
  write: SD_CJY_write,
  ioctl: SD_CJY_ioctl,
};

/* SET_MODULE_OWNER( &my_fops ); */

unsigned int MajorNum=0;
char DriverName[]="SimpleDev";
int init_module(void)
{
    int ret_value;
    printk("<1>Excute init_module(.)\n");
    SET_MODULE_OWNER( &my_fops );
    ret_value=register_chrdev( 0, DriverName, &my_fops );
```

```
    if(ret_value<0)
    {
        printk(KERN_WARNING "driver_name:can't get major number");
        return ret_value;
    }
    else if(MajorNum==0)
    {
        MajorNum=ret_value;
        printk("<1>Get Major Number:%d\n",MajorNum);
        return 0;
    }
}

void cleanup_module(void)
{
    printk("<1>Execute cleanup_module(.)\n");
    if( MajorNum>0 )
    {
        unregister_chrdev( MajorNum, DriverName );
    }
    return;
}

/* Implement of operations functions */
int SD_CJY_open( struct inode *inode, struct file *filp )
{
    int major_num,minor_num;
    printk("<1>Execute SD_CJY_open\n");
    major_num = MAJOR( inode->i_rdev );
    minor_num = MINOR( inode->i_rdev );
    printk("<1>MAJOR NUMBER:%d; MINOR NUMBER:%d\n",major_num,
minor_num);
    return 0;
}

int SD_CJY_close( struct inode* inode, struct file* filp )
{
    printk("<1>Execute SD_CJY_close\n");
```

```
    return 0;
}

ssize_t SD_CJY_read( struct file* filp, char* buf, size_t length, loff_t* floff )
{
    ssize_t length_read=0;
    unsigned int data_length=32;
    char data_in_kernel[]="You Get Data From Simple Device";
    printk("<1>Execute SD_CJY_read(.)\n");
    printk("<1>Data:%s\n",data_in_kernel);
    if( length > data_length )
    {
        return -EFAULT; /* Error */
    }
    if(copy_to_user(buf,data_in_kernel,length)){
        return -EFAULT; /* Error */
    }
    length_read=(ssize_t)length;
    return length_read;
}

#define MaxLength_buf 80
ssize_t SD_CJY_write( struct file* filp, const char* buf, size_t length, loff_t* floff )
{
    ssize_t length_write=0;
    char buf_in_kernel[MaxLength_buf];
    int ii;
    printk("<1>Execute SD_CJY_write(.)\n");
    if(length>MaxLength_buf)
    {
        return -EFAULT; /* Error */
    }
    if(copy_from_user( buf_in_kernel, buf, length ))
    {
        return -EFAULT; /* Error */
    }
    length_write=length;
    *floff+=length;
```

```
    printk("<1>Data writen in Device:");
    for( ii=0; ii<length; ii++ )
    {
        printk("%c",buf_in_kernel[ii]);
    }
    printk("\n");

    return length_write;
}

int SD_CJY_ioctl(struct inode* inode, struct file* filp, unsigned int command_code,
unsigned long command_parameter)
{
    printk("<1>Execute SD_CJY_ioctl\n");
    printk("<1>Get Command Code:%d\n",command_code);
    printk("<1>Get Command Parameter:%ld\n",command_parameter);
    return 0;
}
```

Makefile:

```
CC=gcc
CFLAGS = -O2
IPATH = -I/usr/src/linux/include
SOURCE = SimpleDeviceModule.c
OBJ = SimpleDeviceModule.o

all:
        $(CC) $(CFLAGS) $(IPATH) -c $(SOURCE) -o $(OBJ)
```

Load script file:

```
# !/bin/sh
DeviceModuleName="SimpleDeviceModule.o"
DeviceRegistedName="SimpleDev"
minor="0"$1;
NodeFileName="$DeviceRegistedName"$minor
```

```
group="milton"
mode="666"

# Load Module
/sbin/lsmod
/sbin/insmod $DeviceModuleName
/sbin/lsmod

# Find Major Number
major=`awk "/$DeviceRegistedName/ {print \\$1}" /proc/devices`

# Show Major Number on Console
echo 'major number:'$major

# Move unnecessary Node File
rm -f /dev/${DeviceRegistedName}

# Create Node File
mknod /dev/${NodeFileName} c $major $minor

# Assign correction group and access right
chgrp $group /dev/${NodeFileName}
chmod $mode /dev/${NodeFileName}

# Show the Node File
ls /dev/${NodeFileName} -l
```

Unload script file:

```
# !/bin/sh
NodeFileName="SimpleDev0"$1
ModuleName="SimpleDeviceModule"

# Remove Module
/sbin/lsmod
/sbin/rmmod $ModuleName
/sbin/lsmod
```

```
# Remove Node File
ls -l /dev/${NodeFileName}
rm -f /dev/${NodeFileName}
ls -l /dev/${NodeFileName}
```

使用上述驅動程式的應用程式 App01.c:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main( int argc, char argv[] )
{
   int devfp; /* Device Node File pointer */
   char NodeFileName[]="//dev//SimpleDev0";
   char Buffer[100];
   char Data[]="This data come from App01";
   int count=32,count_read=0;
   int ii;
   int command_id;
   long command_para;

   /* Open the device */
   devfp = open( NodeFileName,O_RDWR );
   if( devfp <= 0 )
   {
      printf("Node File Opne Fail: The %s doesn't exist\n",NodeFileName );
      return -1; /* Error */
   }

   /* Read data from device */
   count_read = (int)read(devfp, (void*)Buffer, (size_t)count );
   /* Show what we read */
   for( ii=0; ii<count_read; ii++ )
   {
      printf("%c",Buffer[ii]);
```

```
    }
    printf("\n");

    /* Prepare data for writing to device */
    count=26;

    /* Write data to device */
    count_read = (int)write(devfp, (void*)Data, (size_t)count );
    if( count_read!=count )
    printf("Fail Sending data to device\n");

    command_id=10;
    command_para=31;
    /* Set IO control command to device */
    if(ioctl( devfp, command_id, command_para )<0)
    {
       printf("Fail Senting Command to device\n");
    }

    /* Close the device */
    close( devfp );

    /* end main */
    return 0;
}
```