Lab 10 Linux Drivers

實驗目的

利用 Linux 來建立 Embedded system 已經是非常常見的應用。在不同的系統應用上面經常會有不同的硬體如 LCD、Key Pad 等，必須透過不同的 I/O 操作才能夠控制。通常這些必須直接與硬體溝通的程式，都會寫成 driver 的型態載入作業系統。透過該 driver 提供一組標準的介面存取硬體，如此使用者的程式便不需要直接與硬體溝通，當更換了不同的硬體，也只需載入不同的 driver，不用重寫上層的應用程式，也就是說 driver 為 kernel space 和 user space 間的 interface。本章的目的是讓大家對 driver 能夠有一個初步的認識，並且用簡單的例子描述一個基本的 driver 如何撰寫。在讀完本章之後，便能具備設計和修改 driver 的基本功力了。

實驗器材

- PC x 1
  - Requirement: any modern PC will do.
  - Purpose: To work as a host workstation on which we will run Linux to create and build a customized driver.

實驗所需軟體

- PC
  - Linux
  - Editor (such as vi and emacs)
  - gcc compiler

Table of Contents

第一次寫 Linux Driver 就上手

1.簡介

Linux 將 driver 分為三種型態，分別是字元、區塊和網路設備，本章將以最常用的字元裝置當作例子。一般而言字元裝置可當作一般檔案存取，包含基本的 open、close、I/O control、read 和 write。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。我們將這幾項服務列成 event 來看，並且一步步引導大家來實踐這些 event。

Initial module

當 driver 被載入之後第一個被呼叫的函式，類似一般 C 語言中的 main function，在此 function 中向系統註冊為字元 device 和所提供的服務

Open device

當我們的 device 被 fopen 之類的函式開啟時所執行的對應處理函式

Close device

使用者程式關閉我們的 device 時執行的對應處理函式

I/O control

使用者可透過 ioctl 命令設定 device 的一些參數

Read device

當程式從我們的 device 讀取資料時對應的處理函式

Write device

當程式對我們的 device 寫入資料時對應的處理函式

Remove module

當 driver 被移除時所執行的處理函式，必須對系統取消註冊 device

2. 如何寫一個 driver

建立一個基本的 driver

最簡單的 driver 架構非常簡單，只需要兩行：

demo.c

```
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
```

Makefile 如下：

```
CC=arm-unknown-linux-gnu-gcc
# 與編 kernel image 時用的 cross compiler 相同

obj-m := demo.o

all:
	make -C /[PATH]/pxa270/linux M=$(PWD) modules
clean:
	make -C /[PATH]/pxa270/linux M=$(PWD) clean
```

Compile driver:

注意所使用的 cross compiler 必需和編 kernel image 時用的 cross compiler 相同(例：arm-unknown-linux-gnu-gcc)；make 的路徑參數為 kernel 的 source 所在(例：[path]/pxa270/linux)。

使用 make 指令編譯。此時會輸出許多檔案，其中有一個叫作 demo.ko 的檔案，恭喜，這就是我們的第一個 driver 了

如果你 make 的時候失敗了，有可能是因為你有編 kernel 失敗遺留下來的東西，請重新完整地編過 kernel 一次

Install driver:

要將 driver 安裝到系統之中，只需要下一個命令：

insmod demo.ko

因為我們的 driver 中並沒有任何 initial，一般而言都會安裝成功。此時只要再下

lsmod

此命令會列出所有系統中的 driver，在列表中可以看到我們剛剛安裝的 driver

（若出現 wrong version magic 問題，請確認你編譯 driver 用的 cross-compiler 與編 kernel 所使用的 compiler 相同。）

Remove driver:

當我們已經不需要該 driver 時，若是要節省系統資源可以將其釋放，只需要下一個命令即可：

rmmod demo

注意 busybox 預設編入的指令沒有 insmod 和 rmmod，請記得在編 file system 時將與 module 相關的指令編入。例：

Linux Module Utilities →

[*] insmod

[*] Support version 2.6.x Linux kernels

[*] lsmod

[*] modprobe

[*] rmmod

[*] Support tainted module checking with new kernels

加入 Initial Module 和 Remove Module

上面的例子示範了如何建立一個沒有任何功能的 driver，相信大家都覺得非常的簡單愉快。為了排遣各位的無聊，接下了馬上為大家介紹如何加上 initial module 和 remove module，我們將 demo.c 修改成下面的程式：

demo.c

```
//Includes essential headers
#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 會再執行 rmmod 指令時被呼叫。請大家再重新 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 dmsg 命令或者是到 /var/log/syslog 中察看。

加入 open、close、I/O control、read 和 write

經過了上面的步驟之後，只要再加上 device 註冊並提供基本的檔案操作功能，我們的 driver 大致上就大功告成。下面示範最一個簡單的範例

```
#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;
}
```

```
int drv_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
          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,
          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);
```

上面的程式提供了一個字元裝置最基本的架構。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 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。另外在 remove module 的時候必須呼叫 unregister_chrdev 取消裝置註冊，以免系統產生異常喔。

建立裝置檔案

當 driver 已經完成，要如何使用程式來開啟該裝置呢？在 Linux 中，大部分的 device 都以檔案的型態存在/dev/目錄之下。現在我們必須建立一個裝置節點的檔案，透過以下指令來建立對應的裝置節點：

mknod /dev/demo c 60 0

其中/dev/demo 是裝置名稱，c 代表字元裝置，60 代表主要版本，0 代表次要版本。當 driver 在向系統註冊的時候也必須用同樣的型態、名稱和主要版本註冊，以免失敗。

3. 使用你的 driver

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：

test.c

```
#include <stdio.h>
int main()
{
        char buf[512];
        FILE *fp = fopen("/dev/demo", "w+");
        if(fp == NULL) {
                printf("can't open device!\n");
                return 0;
        }
        fread(buf, sizeof(buf), 1, fp);
        fwrite(buf, sizeof(buf), 1, fp);
        fclose(fp);
        return 0;
}
```

編譯(arm-linux-gcc --static -gdwarf-2 test.c)此測試程式，並把它放到目標板上執行後，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)。

4. 和 I/O 溝通

Driver 主要的目的在於提供一個和硬體溝通的介面給應用程式。目前為止我們只描述了一個 driver 的模型要如何建構，但是並沒有真正的和硬體有任何通訊。因為與硬體實際的通訊會因為不同的硬體規格和接腳位置而有所不同，這邊將以講解概念為主，實作上請同學自行參考硬體的 specs。

I/O 主要分為 memory mapped I/O 和 port I/O，使用哪一種會因系統而異(80x86 系列大部分是使用 port I/O，而許多 embedded system 則是採用 memory mapped I/O)，以下將對於兩種不同的使用模式作個簡單的介紹：

Memory mapped I/O

使用和存取 memory 相同的指令進行 I/O，將 I/O port 當作 memory address 的一部份稱為 memory mapped I/O。假設我們的 device 有一個 8 bit 的 I/O port 接到我們的系統，記憶體位址在 0x10000，我們的程式可以透過下列範例所介紹的方式進行讀寫

```
#define DATA_PORT (*(volatile char*)(0x10000))
char read_b() {
   return DATA_PORT;
}
void write_b(char data) {
   DATA_PORT = data;
}
```

Port I/O

I/O 有獨立的 I/O address 並且使用不同於存取記憶體的命令操作，稱為 port I/O。當系統所使用的 I/O 為 Port I/O，我們可以使用 inb 和 outb 等系統函式對 I/O port 進行操作，不過在那之前必須先保留 I/O 位址給 driver，下列程式為確認並保留 PC 系統上的 parallel port 給我們的程式：

```
  /* Registering port */
  int  port = check_region(0x378, 1);
  if (port) {
```

```
    printk("<1>parlelport: cannot reserve 0x378\n");
    return -1;
  }
  request_region(0x378, 1, "demo");
```

當 driver 解除安裝時需要釋放 I/O 位址

```
  release_region(0x378,1);
```

接下來，就可以在程式中利用 inb 和 outb 對裝置進行 I/O 囉

```
char read_b() {
   return inb(0x378);
}
void write_b(char data) {
   outb(data, 0x378);
}
```

參考文獻：

http://www.freesoftwaremagazine.com/articles/drivers_linux

教育部顧問式嵌入式軟體聯盟
實驗模組整合與單一平台建置計畫

實驗模組名稱:	第一次寫 Linux Driver 就上手
開發教師:	施吉昇 (cshih@csie.ntu.edu.tw)
開發學生:	張原豪
	陳榮彰
學校系所:	台灣大學資訊網路與多媒體研究所/資訊工程學系
聯絡電話:	02-33664927
聯絡地址:	台北市羅斯福路四段一號
	資訊工程系（德田館）523 室
繳交日期:	2006 年 5 月 30 日
實驗平台:	Intel Xscale 270 and DSP Dual core
實驗主軸:	Device Driver
實驗內容關鍵字:	Linux Driver