*/proc* is a virtual file system which only occupies the memory space.

*# cat /proc/partitions:* check the disk name.

*# cat /proc/interrupts*: tell you the devices and their relative IRQ number and how many of each there has been. You can check if a specified interrupt working properly, and count the times of interrupts which are triggered. After you insert the modules, you can see the interrupt name you register by calling *request\_irq()*. While AP is running to call the driver, you can see the times of interrupt are increasing.

Linux核心固定週期會發出timer interrupt (IRQ 0)，HZ定義每秒幾次timer interrupts。HZ可在編譯核心時設定：  
*$ make menuconfig  
Processor type and features ---> Timer frequency (250 HZ) --->*

其中HZ可設定100、250、300或1000。觀察*/proc/interrupt*的timer中斷次數，並於一秒後再次觀察其值。理論上兩者應該相差250左右。Tick是HZ的倒數，意即timer interrupt每發生一次中斷的時間。

*$ cat /proc/interrupts | grep timer && sleep 1 && cat /proc/interrupts | grep timer*  
0: 9309306 IO-APIC-edge timer  
0: 9309562 IO-APIC-edge timer

上四個欄位分別為中斷號碼、CPU中斷次數、PIC與裝置名稱。timer interrupt會做哪些事情? (1)更新時間、日期與系統從開機至目前經過多少時間。(2)更新系統資源使用率統計。(3)檢查正在執行的程序是否超過分配的執行時間額度。如果是，則侵佔(preempt)該程序以利執行其它等待執行的程序。(4)檢查軟體時間器(Software timer，如alarm系統呼叫)跟時間延遲函式(Delay function)的延遲時間是否已經超過。Tick是HZ的倒數，即timer interrupt每發生一次中斷的時間。  
*Jiffies為Linux核心變數(32位元變數，unsigned long)*，它被用來紀錄系統自開幾以來，已經過多少的tick。每次timer interrupt，Jiffies會被加一。Jiffies於系統開機時並非初始化成零，而是被設為-300\*HZ (*arch/i386/kernel/time.c*)，即代表系統於開機五分鐘後，jiffies便會溢位。80x86架構定義一個與jiffies相關的變數jiffies\_64，此變數64位元，jiffies被對應至jiffies\_64最低的32位元。因此經由jiffies\_64可以完全不理會溢位的問題便能取得jiffies。

# *cat /proc/cpuinfo*: the CPU information.

*# cat /proc/devices*: the device list.

*# cat /proc/ioports*: the IO port. Some default IO ports are defined by BIOS. Use *request\_region()/release\_region()* to register/unregister IO ports and the information is from */proc/ioports*:

*#define SPI\_BASE\_ADDR 0xf00*

*request\_region(SPI\_BASE\_ADDR, 4, “spi”);*

Then you can see in */proc/ioports*:

*0xf00-0xf04 : “spi”*.

*# cat /proc/module*: the modules installed in this system.

*# cat /proc/version*: the version of Linux kernel and gcc. (*uname –r* can only shows the kernel version.)

*# cat /proc/cmdline*: the kernel command line.

The following command will only show the error message of level 2: *# make –j4 2> error.txt*.

To use SSH to remote log-in a Linux system, type the following command: *# ssh –X price@192.168.1.120*.

To clear the kernel message, use *dmesg –c*.

Ubuntu uses */bin/bash* shell and Android uses */system/bin/sh*. While writing shell script, mind to use the proper shell in different OS, otherwise the script can’t run properly.

Before building the new kernel, mind to clean the old files first.

Use *modprobe* to install the Linux driver which is marked as “m” in *.config* and this command can be executed anywhere in Linux.

Before building kernel, mind that use *make mrproper* or *make clean* to clean the old files. Usually, *make mrproper* is only used for the first time because it remove all the files including *.config* and *Module.symvrs* in each module folder. Use *make* to build Linux kernel, but the modules which is marked as “M” in *menuconfig* are not be compiled. Then use *make modules* to build these modules and finally use *make modules\_install* to install these modules in */lib/modules/<Linux kernel>/kernel…*

If the build machine is 64-bit one, make sure of adding *ARCH=i386* before building kernel image. You can check *CONFIG\_X86\_32=y* and *CONFIG\_X86\_64* is not set in the *.config* file. Make menu configuration is similar:

# *make ARCH=i386 menuconfig*.

以 *lspci* 列出所有接在 PCI bus上的 device，不論是否已有 driver 驅動。如下列：

00:00.0 Host bridge: VIA Technologies, Inc. VT8377 [KT400/KT600 AGP] Host Bridge

00:0b.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)

第一個欄位，如00:0b.0代表PCI bus ID。在*/sys/bus/pci/devices/*下可見到bus ID，只不過前頭再加上0000:的目錄。進入某個device的PCI bus目錄，當中有重要的兩個檔案：vendor及device，分別表示這個PCI device的vendor ID及device ID。查看這兩個檔案以取得到需要的 vendor/device ID。

$ *cat /sys/bus/pci/devices/0000:00:0b.0/vendor*  0x8086

$ *cat /sys/bus/pci/devices/0000:00:0b.0/device*   0x100e

USB Devices

PCI devices 類似，差別只是在改找USB bus ID，也就是VID (vendor ID)

PID(product ID)。利用 *lsusb*找出USB裝置的VID/PID。可觀察USB裝置插入/移除後，那個device出現又消失。lsusb 得到的資訊如下：

$ /usr/sbin/lsusb | grep -v 0000:0000 Bus 004 Device 003: ID 157e:300d

其中ID後的兩組數字分別就是 USB VID/PID。From different Linux version, the usb information may locate in different path. Three possible paths are: (1) */proc/bus/usb*. (2) */dev/bus/usb*. (3) */sys/bus/usb/devices*.

The configuration item *CONFIG\_USB\_DEVICEFS* shows USB device filesystem. If you say Y here, you will get a file */proc/bus/usb/devices* which lists the devices currently connected to your USB bus or busses, these nodes now lived in */dev/bus/usb* and are used by *libusb*.

The usbfs filesystem for USB devices is traditionally mounted at */proc/bus/usb*. In many modern systems the usbfs filsystem isn't used at all. Instead USB device nodes are created under */dev/usb/* or someplace similar. Note if */proc/bus/usb* appears empty, and a host controller driver has been linked, then you need to mount the filesystem. Issue the command (as root): *mount -t usbfs none /proc/bus/usb*. You can then issue `*cat /proc/bus/usb/devices*` to extract USB device information, and user mode drivers can use *usbfs* to interact with USB devices.

Each connected USB device has one /*proc/bus/usb/BBB/DDD* file. The BBB indicates the bus number. The DDD indicates the device address on that bus. Both of these numbers are assigned sequentially, and can be reused, so you can't rely on them for stable access to devices. These files may also be used to write user-level drivers for the USB devices. You would open the */proc/bus/usb/BBB/DDD* file *read/write*, read its descriptors to make sure it's the device you expect, and then bind to an interface (or perhaps several) using an *ioctl* call. Note that since by default these BBB/DDD files are writable only by root, only root can write such user mode drivers.

To search for the files and edit them: *#find . –name \*.reg | xargs gedit*.

Linux kernel source code exists in */usr/src/linux-headers-2.6.38.33-generic/*. The binary kernel after compiling is in */lib/modules/2.6.38.33/kernel/* and you can see some symbolic links to the real source code.

*dd*可從標準輸入裝置(或檔案讀取資料)經過指定的格式來轉換資料，再輸出至檔案、裝置或標準輸出。This command can be categories into three levels: disk, partition and file. The copy can be (1) disk->disk. (2) partition->partition. (3) disk, partition, file->file. (4) file->disk, partition.

Disk->disk:硬碟整個到另一顆硬碟備分的指令：*＃dd if=/dev/hda of=/dev/hdb*. (硬碟對拷，格式必需一致，因為不同硬碟有不同的Sector和Cylinder的大小，拷備到目地槽可能會發生檔案讀取錯誤之問題；若為Flash或CF卡則無此問題)

Partition->partition:一個Partition到另一Partition備分的指令：*＃dd if=/dev/sda1 of=/dev/sdb1*.

In the disk, the first sector is MBR and FAT. This information belongs to entire disk. Then there are data in different partitions which are stored in order. If you use *dd* command to copy data between partitions, the data in MBR and FAT will not be copied.

備份: *dd if=/dev/hda1 of=/home/backup\_system/file\_name.* 還原: *dd if=/home/backup\_system/file\_name of=/dev/hda1* (如果要備分的是partition，不能將"要備分的分割區"放在同一個partition內，如：# dd if=/dev/hda1 of=/dev/hda2/copy\_datas).

Two drivers work together so that TREK550 CAN device can work properly. *interrupt\_driver.ko* calls the CAN sdk provided by chip vendor to communication with SPI directly and exports another set of sdk used by *canbus.ko*. Since the driver has to access the SPI which means it is required to register the IO space in x86 CPU. Use *request\_region()* and *release\_region()* to register/unregister the space which is used by CAN driver (Type #*cat /proc/ioports* so that you the address named “spi”). In *canbus.ko,* two things need to be done in *module\_init()*: (1) Register the device number. (2) Register the IRQ (using *request\_irq()*, type #cat */proc/interrupts* so that you can see the IRQ named “can\_spi”). The opposite things are called in *module\_exit()*. Then, initialize the CAN controller so that it starts to access CAN data.

Linux 2.6.19以後，中斷處理interface有改變。*IRQF\_xxx*型中斷標誌取代了*SA\_xxx*型中斷標誌。在較早的内核中應該使用*SA\_INTERRUPT*而不是*IRQF\_DISABLED*來將中斷處理標記為快中斷處理。

*request\_irq(IRQ\_NUM, corgikbd\_interrupt, IRQF\_DISABLED | IRQF\_TRIGGER\_RISING, "corgikbd", corgikbd);*

Linux device driver中名為“interrupt handler”的routine負責處理實體的硬體中斷。當裝置中斷被觸發時，interrupt handler便會執行回應該中斷的請求。Interrupt handler執行於interrupt mode，無process context資訊。在interrupt mode：1. 沒有process context的關係，因此無法存取user space。2. 無法存取*current*巨集。*current*是指向自己的kernel symbol。3. 不能呼叫scheduler做排程，也不能sleeping waiting。由於down/up的semaphore API是sleeping waiting的版本，因此在interrupt mode須改用spinlock，以busy waiting的方式做wait operation。Linux driver安裝interrupt handler是呼叫*int request\_irq( unsigned int irq, void (\*handler)(int, void \*, struct pt\_regs \*), unsigned long irqflags, const char \*devname, void \*dev\_id);*

呼叫*request\_irq()*安裝interrupt handler的位置為：1. *init\_module()*. 2. *fops->open*. 共用中斷只能在*fops->open*裡呼叫*request\_irq()*。在 *fops->open*裡請求IRQ，相對的要在*fops->release*裡呼叫*free\_irq()*將佔用的IRQ釋放。

The *DECLARE\_WAIT\_QUEUE\_HEAD* macro declares a wait queue to make the process sleep. The process goes to sleep because of *wait\_event\_interruptible()*. In CAN driver, when interrupt occurs, the ISR is called which get the CAN data from the queue. Then use *wake\_up\_interruptible()* to invoke the process. The pseudo-codes are shown below:

*DECLARE\_WAIT\_QUEUE\_HEAD(WaitQueue);*

*INT Flag = 0;*

*static int adv\_canbus\_ioctl(…){*

*if(cmd == TREK550\_SPI\_WAIT\_EVENT){*

*wait\_event\_interruptible(WaitQueue, Flag != 0);*

*Flag = 0;*

*}*

*}*

*static irqreturn\_t adv\_canbus\_isr(int irq, void\* arg){*

*…*

*Flag = 1;*

*Wake\_up\_interruptible(&WaitQueue);*

*}*

kernel 2.6.35及之前*struct file\_operations* 共有3个*ioctl*：*ioctl*, *unlocked\_ioctl*和*compat\_ioctl*。现在只有*unlocked\_ioctl*和*compat\_ioctl*。kernel 2.6.36中删除*struct file\_operations*中的ioctl函数指针，取而代之的是unlocked\_ioctl。改变后参数中少了*inode*，用户程序中的ioctl对应的系统调用接口没有变化，所以用户程序不需要改变，一切都交给内核处理。*unlocked\_ioctl*取代*ioctl*。Caution: since the difference between *unlocked\_ioctl()* and *ioctl()* is just the *inode*, so if you use *file\_operations::unlocked\_ioctl()* to point to *ioctl()* in the driver, the compiler can still pass, but the arguments are totally incorrect. *ioctl()* is still used in user mode.

如果已经定义*KERNELRELEASE*，则说明是从内核整体编译的Make中调用。在内核的Makefile中L350行有关于KERNELRELEASE定义：*KERNELRELEASE = $(shell cat include/config/kernel.release 2> /dev/null)*  
ifneq ($(KERNELRELEASE), ) # Using Linux building system to compile.obj-m := hello.oelse

#否则，是直接从命令行调用，这时需要调用内核构造系统，对$(shell uname -r)会得到Linux系统版本相关信息KERNELDIR ?= /lib/modules/$(shell uname -r)/buildPWD := $(shell pwd)Build:    $(MAKE) -C $(KERNELDIR) M=$(PWD) modulesendif

Use ldd to check the executable/library needs which SO file, like ldd ./libSUSI\_IMC\_COMMUN.so.1.0.0. If a so file needs another so file in the same path, but fails to find it, it is because you don’t set the local variable LD\_LIBRARY\_PATH.

The structure definition may be different in different Linux version so that you can get the error: “warning: initialization from incompatible pointer type”. When it happens, check the version of kernel and gcc. In Linux, the compilation of APs and APIs are relative to the version of gcc and the distribution of Linux (Like Ubuntu, red hat). The compilation of drivers are relative to the version of kernel and gcc, and the distribution of Linux.

步骤一：在被依赖的模块B中导出要用的符号,如：

*int cat9555\_state\_get\_inp(void){ ................ }*

*EXPORT\_SYMBOL(cat9555\_state\_get\_inp);*

步骤二：编译模块B，当前目录下生成Module.symvers，将其复制到模块A的目录下。*Module.symvers*内容如下：  
*0x20473c2b cat9555\_state\_get\_inp /home/project/MeterRead/gpio/gpio EXPORT\_SYMBOL*

步骤三：在模块A的文件中引用导出的符号。

*extern cat9555\_state\_get\_inp(void);*      //声明引用的符号是外部变量。

*int f(){cat9555\_state\_get\_inp();*  //在这里引用导出的符号}

编译A模块。我如果没有步骤二的话，插入A模块时会出现以下错误(找不到符号)：

*gprsiodrv: no symbol version for cat9555\_state\_get\_inp  
gprsiodrv: Unknown symbol cat9555\_state\_get\_inp  
insmod: cannot insert 'gprsiodrv.ko': unknown symbol in module*

但是有人不用步骤二也可以，猜测与编译器有关。

If you use the console windows to invoke the application and then hope to run in the background, then use Ctrl+Z to unhook the application and type: *#bg*, the application can switch from foreground to background.

*#cat /proc/tty/driver/serial* lists the usage statistics and status of each of the serial *tty* lines like:

*0: uart:16550A port:3F8 irq:4 tx:0 rx:0  (2 wires)*

*1: uart:16550A port:2F8 irq:3 tx:6 rx:11245 bark:12 RTS|DTR|DST  (6 wires).*

16550A is the COM port IC and replaces with 8250. You can read the times of sending and receiving data from tx and rx.

*# dmesg | grep tty* shows the mapping of COM port number:

Serial8250: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A

Serial8250: ttyS1 at I/O 0x2e8 (irq = 3) is a 16550A

In Linux, if you would like to find the relation between COM\* (Defined by BIOS) with ttyS\* (Defined by Linux kernel), check the “Super IO Configuration” page in BIOS to find every COM port and its corresponding IO address and IRQ number. Then you can type *#dmesg | grep ttyS\** in Linux kernel so that you can get every *ttyS\** device node and its corresponding IO address and IRQ number. Find the mapping of IO address and IRQ number between two lists then you can find the relation between *COM\** and *ttyS\**.

The default value of setting maximum number of 8250/16550 serial ports is 4. To set the value to 6, modify *CONFIG\_SERIAL\_8250\_NR\_UARTS=6* and *CONFIG\_SERIAL\_8250\_RUNTIME\_UARTS=6.*

守护进程是脱离终端的，他的信息无法直接输出到标准输出和错误输出设备，Linux系统中提供了*syslog()*系统调用。

*#include <syslog.h>*

*void main(void) {*

*openlog("slog", LOG\_PID|LOG\_CONS, LOG\_USER);*

*syslog(LOG\_INFO, "A different kind of Hello world ... ");*

*closelog();*

*}*

You can use the command: *# tail -f /var/log/messages* to show the dynamic kernel message.

Linux块设备操作可分为两类：1. C标准库中的*fopen/fread/fwrite* (buffered I/O)。I/O path為: Application<->Library Buffer<->OS Cache<->File System/Volume Manager<->Device。library buffer是标准库提供的用户空间的buffer，通过setvbuf改变大小。2. Linux的系统调用*open/read/write* (non-buffered I/O)。I/O path為: Application<-> OS Cache <->File System/Volume Manager<->Device。

通过设置*open*的*O\_DIRECT*标志来实现Direct I/O(或叫Raw I/O)，即绕过OS Cache，直接读取Device，等于将OS cache换成自己管理的cache。如果是大量随机写入操作，*O\_DIRECT*会提升效率。但是顺序写入和读取效率都会降低。字詞轉換是中文維基的一項自動轉換，目的是通過電腦程式自動消除繁簡、地區詞等不同用字模式的差異，以達到閱讀方便。字詞轉換包括全域轉換和手動轉換，本說明所使用的標題轉換和全文轉換技術，都屬於手動轉換。

如果您想對我們的字詞轉換系統提出一些改進建議，或者送出應用面更廣的轉換（[中文維基百科](http://zh.wikipedia.org/wiki/中文维基百科)全站乃至[MediaWiki](http://zh.wikipedia.org/wiki/MediaWiki)軟體），或者報告轉換系統的錯誤，請前往[Wikipedia:](http://zh.wikipedia.org/wiki/Wikipedia:字词转换请求或候选)字詞轉換請求或候選發表您的意見。Use *lsmod* to check which drivers are installed in the machine.

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/*.

If you add “*-soname*” attribute while building the SO file, the SO file or its symbolic link must exist in the compiling time. For example:

In library: *gcc -shared -Wl,-soname,libtest.so -o libtest.so.1.0.0*

In app: *gcc testdemo.c -o testdemo -Wl,--start-group libtest.so.1.0.0 --end-group*

Mind to create symbolic link: *ln –s libtest.so.1.0.0 libtest.so.1* otherwise the error occurs while running.

Use *#dpgk –i package\_file.deb* to install a *.deb* file. To Uninstall a *.deb* file: *#dpkg –r package\_name*. Find the version of a package installed: *#dpkg –l package\_name*. List all installed files with version: *#dpkg –l*.

Before accessing the IO port, call *ioperm()* or *iopl()* to tell the system to provide the access right, otherwise, the segmentation faults occur. If the program exploits *inb(), outb()*, ...,etc, to access the ports, mind adding *–O* attribute to turn on the compiler optimization.

In Linux, there are lots of files whose names are the same but locating in different path. Take *io.h* for example, in user mode, it exists in */usr/lib/sys*, but in *${LinuxKernl}/include/asm-generic* in kernel mode.

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*. Mind that there are also some other files like *uevent*, *subsystem* created concurrently. Mind that the third argument of *device\_create()* (<http://hi.baidu.com/sdqdshixin/item/bb833382c7521aeae596e0c3> for more detailed info about *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, we often define the data structure 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，需要使用*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.

在x86架構裡，主機開機第一個被讀取的地方是BIOS(Basic Input Output System)，BIOS裡記錄主機板晶片組與相關設定，如CPU與周邊設備的溝通時脈、開機裝置的搜尋順序、硬碟大小與類型、系統時間、各周邊匯流排是否啟動Plug and Play(PnP, 隨插即用裝置)、各周邊設備的I/O位址、以及與CPU溝通的IRQ岔斷等等。之後會進行開機裝置的資料讀取(MBR相關的任務開始)。BIOS瞭解主機硬體相關資訊後，主機會開始由儲存媒體載入作業系統，系統會去第一個開機裝置上進行開機程序。開機流程讀到硬碟的過程中，第一個要讀取的是該硬碟的主要開機磁區(Master Boot Record, MBR)，系統可由MBR所安裝的開機管理程式(boot loader)執行核心辨識的工作。每顆硬碟的第一個磁區稱為MBR，若主機上有兩顆硬碟，系統會看BIOS的設定。『系統的 MBR』指的是第一個開機裝置的MBR！想載入Linux核心，得用支援Linux file system的boot loader，目前主流的grub開機管理程式，可支援Linux及Windows相關的核心系統！因Windows

Linux的檔案格式不一樣！為了載入系統核心，必須安裝認識OS的loader，而Linux的loader (lilo或grub)認識windows的核心檔案，但Windows的loader卻不認識Linux的核心檔案，因此Windows提供的loader不能作為多重開機的設定loader。藉由boot loader的管理而開始讀取核心檔案後，接下來Linux會將核心解壓縮到主記憶體中，並利用核心的功能開始測試與驅動各個周邊裝置，包括儲存裝置、CPU、網路卡、音效卡等。一般核心檔案會被放置到*/boot*裡，並取名為*/boot/vmlinuz*！

作業系統核心必須要認識磁碟檔案系統才能讀取裡面的資料，必須要有boot loader才有辦法載入Linux的核心(kernel)！在載入核心的過程當中，系統只會『掛載根目錄』而已，而且是以唯讀的方式掛載。為了讓某些功能可以用檔案的方式來讀取，有的系統在開機時會製作虛擬硬碟(RAM Disk)來輔助，即initrd和linuxrc。利用boot loader的功能，可在載入核心時一起載入initrd的映象檔(*/boot/initrd-xxxx.img*)，Linux系統會主動以initrd來進行虛擬硬碟的建置，並利用linuxrc(包含在initrd的映象檔內)的功能進行載入模組的動作。在核心驅動周邊硬體工作完成後initrd建立的虛擬磁碟就會被移除！不過initrd並非必要。在核心完整的載入後主機就開始運作，接下來開始執行系統的第一支程式：*init*。

核心一般是壓縮檔，使用核心前要解壓縮才能載入主記憶體。目前的核心都具有『可讀取模組化驅動程式(modules)能！該程式可能由硬體開發廠商提供或核心就支援～不過較新的硬體，通常由硬體開發商提供驅動程式！核心與核心模組放在哪？

核心：/boot/vmlinuz 或/boot/vmlinuz-version；核心解壓縮所需RAM Disk：/boot/initrd (/boot/initrd-version)；核心模組：/lib/modules/version/kernel或 /lib/modules/`uname -r`/kernel；核心原始碼：/usr/src/linux (要安裝才會有！預設不安裝！)

To establish a multi-OS x86 system, Windows should be installed first and then Linux. The steps are as followed: (1) In Windows, use disk management tool (like partition master) to re-partition the hard drives and leave some space as “unallocated”. (2) While installing Linux, create the disk partition manually based on the “unallocated” space. Mind that there are at least two partitions: */* and *SWAP*.

To add the touch screen driver in HIT, add the following codes in *linux/kernel/drivers/input/touchscreen/kconfig* as below:

config TOUCHSCREEN\_USB\_PENMOUNT

default y

bool “Dialogue PenMount tablet device support” if EMBEDDED

depends on TOUCHSCREEN\_USB\_COMPOSITE

It shows that *TOUCHSCREEN\_USB\_PENMOUNT* depends on *TOUCHSCREEN\_USB\_COMPOSITE* which means if you set *TOUCHSCREEN\_USB\_COMPOSITE=y* then you can see *TOUCHSCREEN\_USB\_PENMOUNT=y* in *.config*. Otherwise, *TOUCHSCREEN\_USB\_PENMOUNT* disappears when *TOUCHSCREEN\_USB\_COMPOSITE* is not set. Add the implementation of touch panel in *linux/kernel/drivers/input/touchscreen/usbtouchscreen.c*. Besides, define vendor and device ID for PenMount in *linux/kernel/drivers/hid-ids.h*. Set *TOUCHSCREEN\_USB\_PENMOUNT=y* in *android/device/…/xxx\_defconfig*.

Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory. Both are called I/O memory because the difference between registers and memory is transparent to software.

I/O device透過I/O port存取與控制。

I/O device的存取變成記憶體存取。使用者存取I/O裝置就會和CPU的記憶體存取一樣。Linux提供I/O port存取介面：*inb()*, *outb(),...*。若存取I/O memory，則改用*readb()*, *writeb()*。

I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus, and implementing device registers that behave like I/O ports. Depending on the computer platform and bus being used, I/O memory may or may not be accessed through page tables. When access passes though page tables, the kernel must first arrange for the physical address to be visible from your driver, and this usually means that you must call *ioremap**() before doing any I/O. If no page tables are needed, I/O memory locations look pretty much like I/O ports, and you can just read and write to them using proper wrapper functions.* I/O memory regions must be allocated prior to use. The interface for allocation of memory regions (linux*/ioport.h*) is:

*struct resource \*request\_mem\_region(unsigned long start, unsigned long len, char \*name);*

This function allocates a memory region of *len* bytes, starting at *start*. If all goes well, a non-NULL pointer is returned; otherwise the return value is *NULL*. All I/O memory allocations are listed in */proc/iomem*.

Memory regions should be freed when no longer needed:

*void release\_mem\_region(unsigned long start, unsigned long len);*

Once equipped with *ioremap* (and *iounmap*), a device driver can access any I/O memory address, whether or not it is directly mapped to virtual address space. The functions are called according to the following definition (asm/io.h):

*void \*ioremap(unsigned long phys\_addr, unsigned long size);*

*void iounmap(void \* addr);*

On some platforms, you may get away with using the return value from ioremap() as a pointer. Such use is not portable. The proper way of getting at I/O memory is via a set of functions (*asm/io.h*) provided for that purpose. To read from I/O memory, use one of the following:

*unsigned int ioread8/16/32(void \*addr);*

*addr* should be an address obtained from *ioremap()* (perhaps with an integer offset); the return value is what was read from the given I/O memory. There is a similar set of functions for writing to I/O memory:

*void iowrite8/16/32(u8 value, void \*addr);*

When I/O memory is being used, an older set of functions still work, but their use in new code is discouraged:

*readb(), readw(), readl(), writeb(), writew(), writel()*.

The BIOS base physical address starts from 0xF0000. If the runtime size of BIOS is 64K, the entire range of BIOS in memory is 0xF0000 ~ 0XFFFFF. Use *dmidecode* command to read more BIOS info in Linux.

In old version SUSI library, the library switch the privilege mode from user mode to kernel through *iopl()*, so that it can access the IO port by using *inb()*, *outb()*,…etc. It is not safe enough. In the new version, all the codes relative to IO port access have already migrate into kernel, and are implemented in Linux driver.

The four functions *dlopen(), dlsym(), dlclose(), dlerror()* implement the interface to the dynamic linking loader.

*#include <*[*dlfcn.h*](http://linux.die.net/include/dlfcn.h)*>*

*void \*dlopen(const char \*filename, int flag);*

*char \*dlerror(void);*

*void \*dlsym(void \*handle, const char \*symbol);*

*int dlclose(void \*handle);*

Link with *-ldl*.

The function *dlerror()* returns a human readable string describing the most recent error that occurred from *dlopen()*, *dlsym()* or *dlclose()* since the last call to *dlerror()*. It returns *NULL* if no errors have occurred since initialization or since it was last called.

The function *dlopen()* loads the dynamic library file named by the null-terminated string *filename* and returns an opaque "handle" for the dynamic library. If *filename* is NULL, then the returned handle is for the main program. If *filename* contains a slash ("/"), then it is interpreted as a (relative or absolute) pathname. One of two values must be included in *flag*:

*RTLD\_LAZY*: Perform lazy binding. Only resolve symbols as the code that references them is executed. If the symbol is never referenced, then it is never resolved. (Lazy binding is only performed for function references; references to variables are always immediately bound when the library is loaded.)

*RTLD\_NOW*: If this value is specified, all undefined symbols in the library are resolved before *dlopen()* returns. If this cannot be done, an error is returned.

If the same library is loaded again with *dlopen()*, the same file handle is returned. The dl library maintains reference counts for library handles, so a dynamic library is not deallocated until *dlclose()* has been called on it as many times as *dlopen()* has succeeded on it. If *dlopen()* fails for any reason, it returns NULL. The function *dlsym()* takes a "handle" of a dynamic library returned by *dlopen()* and the null-terminated symbol name, returning the address where that symbol is loaded into memory. If the symbol is not found, in the specified library or any of the libraries that were automatically loaded by *dlopen()* when that library was loaded, *dlsym()* returns NULL. Since the value of the symbol could actually be NULL, the correct way to test for an error is to call *dlerror()* to clear any old error conditions, then call *dlsym()*, and then call *dlerror()* again, saving its return value into a variable, and check whether this saved value is not NULL. The function *dlclose()* decrements the reference count on the dynamic library handle *handle*. If the reference count drops to zero and no other loaded libraries use symbols in it, then the dynamic library is unloaded. The function *dlclose()* returns 0 on success, and nonzero on error. For example, load the math library, and print the cosine of 2.0:

*#include <*[*stdio.h*](http://linux.die.net/include/stdio.h)*>*

*#include <*[*stdlib.h*](http://linux.die.net/include/stdlib.h)*>*

*#include <*[*dlfcn.h*](http://linux.die.net/include/dlfcn.h)*>*

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

*void \*handle;*

*double (\*cosine)(double);*

*char \*error;*

*handle = dlopen("libm.so", RTLD\_LAZY);*

*if (!handle) { fprintf(stderr, "%s\n", dlerror()); exit(EXIT\_FAILURE); }*

*dlerror(); /\* Clear any existing error \*/*

*/\* Writing: cosine = (double (\*)(double)) dlsym(handle, "cos"); would seem more natural, but the C99 standard leaves casting from "void \*" to a function pointer undefined. The assignment used below is the POSIX.1-2003 (Technical Corrigendum 1) workaround; see the Rationale for the POSIX specification of dlsym(). \*/*

*\*(void \*\*) (&cosine) = dlsym(handle, "cos");*

*if ((error = dlerror()) != NULL) { fprintf(stderr, "%s\n", error); exit(EXIT\_FAILURE); }*

*printf("%f\n", (\*cosine)(2.0));*

*dlclose(handle);*

*exit(EXIT\_SUCCESS);*

*}*

If this program were in a file named "foo.c", you would build the program with the following command:

*gcc -rdynamic -o foo foo.c -ldl*

Libraries exporting \_init() and \_fini() will want to be compiled as follows, using *bar.c* as the example name:

*gcc -shared -nostartfiles -o bar bar.c*

One of Linux strong points over Windows is the capability to switch desktops. Several exist, but the two most popular are Gnome and KDE. For Windows users, KDE will have a familiar Windows XP feel, while Gnome may seem dull.

If you have been using Ubuntu, which uses Gnome as the default desktop, or Kubuntu which uses KDE as the desktop, and have been wondering what the other desktop looks like, you can easily install KDE or Gnome and switch back and forth before logging on to Ubuntu. To install KDE or GNOME, refer to the following tow website:

<http://www.omgubuntu.co.uk/2011/10/gnome-shell-ubuntu-11-10-guide/>

<http://www.watchingthenet.com/switch-between-gnome-and-kde-desktops-in-ubuntu-or-kubuntu.html>

The command, *grep -R "SmsFilterData" ./dom/sms/ | cut -d ':' -f 1 | sort -u | wc -l*, shows: search for the key word in a designated folder, acquire the first field in each line, sort them and then count the amount.

When doing the make of a program, you may get this error: Relocation R\_X86\_64\_32 against `vtable for Torch::MemoryDataSet' can’t be used when making a shared object; recompile with –fPIC.  
This problem is related with the use of a 64 bits machine, and exploits *CXXFLAGS* instead of *CFLAGS* when using g++ to compile the codes.

* 1. Ubuntu執行shell script為發生unexpected operator? 因為sh xxxxx.xx必須看sh預設是用bash或是dash， Ubuntu下sh默認指向dash。#*pe sh --> /bin/sh* #*ls –l /bin/sh*

到/bin下有"sh -> dash"，原來sh是/bin/dash的鏈接。Ubuntu6.10已將默認的bashshell更換為dash。其表現是/bin/sh鏈接到了/bin/dash而不是傳統的/bin/bash。dash是一輕量化的shell，它速度更快，但功能相比bash少很多，語法嚴格遵守POSIX標準。目前 Ubuntu及Debian都採用dash作為預設的shell。- dash

bash寫法的差異:

<http://www.igigo.net/archives/169>。處理方式將dash轉為bash:

* 1. - 最暴力的方法當然是直接把/bin/sh的軟鏈接改到bash中，如：ln -s /bin/bash /bin/sh
  2. - 或是直接改用bash 執行該script：bash xxxxxx.sh

If you want to interpret $replace, you shouldn’t use single quotes since they prevent variable substitution. Be careful to ensure that ${replace} doesn't have any characters of significance to sed (like / for instance) since it will cause confusion unless escaped.

pax> export replace=987654321

pax> echo X123456789X | sed ‘s/123456789/${replace}/g’

X987654321X

Ever try to cut (or copy) some lines and paste to another place? If you need to count the lines first, then try these to eliminate counting task. Cut and paste: (1) Position the cursor where you want to begin cutting. (2) Press v (or upper case V if you want to cut whole lines). (3) Move the cursor to the end of what you want to cut. (4) Press d. (5) Move to where you would like to paste. (6) Press p to paste after the cursor, or P to paste before. Copy and paste can be performed with the same steps, only pressing y instead of d in step 4. The name of the mark used is related to the operation (d:delete or y:yank).

从命令行执行history後，通常只会显示已执行命令的序号和命令本身。如果想要查看命令历史的时间戳，可以执行：

# export HISTTIMEFORMAT='%F %T '

# history | more

NTP 要另外安裝，才能發揮持續更新調整時間的功能。安裝 NTP，指令為： *$ sudo apt-get install ntp*

編輯時間伺服器組態檔案 ntp.conf。指令為： *$ sudo vi /etc/ntp.conf. ntp.conf* 檔案的內容如下：

* 第1行Server後面填入time.stdtime.gov.tw：server time.stdtime.gov.tw。
* 第2行Server後面填入clock.stdtime.gov.tw：server clock.stdtime.gov.tw。

測試指令如下：*$ sudo ntpdate time.stdtime.gov.tw*

輸入上述指令，回應如下：28 Aug 13:12:09 ntpdate[29439]: adjust time server 220.130.158.52 offset -0.126794 sec

現在Ubuntu Server可以自行管理時間了

* 1. 如果是Ubuntu 64bit需要安裝ia32-libs，才能跑32bit的程式。Ia32 contains runtime libraries for the ia32/i386 architecture, configured for use on an amd64 or ia64 Debian system running a 64-bit kernel.

SSH分客戶端openssh-client和openssh-server 。如果只想登陸別的機器的SSH只需要安裝openssh-client(ubuntu有默認安裝，如果沒有則sudo apt-get install openssh-client)，如果要使本機開放SSH服務就需要安裝openssh-server:   
sudo apt-get install openssh-server ，然後確認sshserver是否啟動了：  
ps -e |grep ssh   
如果看到sshd那說明ssh-server已經啟動了。如果沒有則可以這樣啟動：sudo /etc/init.d/ssh start   
ssh-server配置文件位於/ etc/ssh/sshd\_config，在這裡可以定義SSH的服務端口，默認端口是22，你可以自己定義成其他端口號，如222。然後重啟SSH服務：  
sudo /etc/init.d/ssh stop   
sudo /etc/init.d/ssh start   
然後使用以下方式登陸SSH：  
ssh tuns@192.168.0.100 tuns為192.168.0.100機器上的用戶，需要輸入密碼。  
斷開連接：exit