While compiling the drivers in android, it is sometimes required to check if the kernel supports the some libraries. Take *w83627ehf-gpio.c* for example, the compilation fails if the *GPIOLIB* is not included in kernel.

To find which shell Android uses, you can go to */system/bin*, and type *find / -name ‘\*.sh’* on Android machine.

W83627 is a chip for hardware monitoring and Linux kernel support this driver. Use *make menuconfig* to modify the *.config* (you can also edit *.config* directly) file so that *SENSORS\_W83627HF=y* and *SENSORS\_W83627EHF=y*. The Linux drivers often exist in *${AndroidPath}/kernel/drivers* folder.

Use the command:

*#mount*

You can get all file systems mounting on the OS.

Then you may need to change the attribute of file system to read/write or read-only mode in the following way:

(1) Change the root file system to read/write mode: *mount –o rw,remount –t rootfs rootfs /* (Caution : the first rootfs means the type of root file system, the second rootfs represents the logical partition of root file system).

(2) Change the root file system to read-only mode: *mount –o ro,remount –t rootfs rootfs /*.

By default, only */system* and */data* folders in android root file system are writable, other folders are read-only. To change the attribute, use mount/remount command shown above.

There are two ways for smart card reader to pass data to user space: USB or COM port. If the reader exploits USB for transferring data, open the support of USB file system in kernel shown below:

USB 裝置檔案系統(USB Device File System)是一個虛擬的檔案系統，讓使用者能夠查詢到USB BUS上所有裝置的相關資訊。當USB Device System被掛載後，它會在掛載的目錄下出現一個devices的檔案。該檔案會列舉出所有連接到USB Bus上所有裝置，與其詳細的內容(如裝置類別, 組態設定, 介面, 設定, 端點...等相關資訊).

開啟核心的USB Device File System支援:

   make menuconfig  
   Device Drivers  --->  
            [\*] USB support  --->  
                \*\*\* Miscellaneous USB options \*\*\*                                                                 
                [\*]     USB device filesystem (DEPRECATED)

掛載USB Device File System: *mount -t usbfs none /dev/bus/usb*

In *hit\_w121*, *sch311x\_wdt* is used as watchdog chip. After installing the driver, you can find */dev/watchdog* device node.

The SUSI library accesses this node to implement the watchdog APIs. In *OpenWdtDrv()*, the system calls *open()* to get the handle point to */dev/watchdog*. *ioctl()* is used in other places of watchdog APIs to set configuration and control the chip.

If you would like to change the android sdk version in your APK project, do as below:

(1) Change the value of *android::minSdkVersion* in *AndroidManifest.xml*.

(2) Select the project you have to modify and right-click on it. Choose the property item in popup menu and select the correct Android sdk version.

The USB information is shown */sys/bus/usb/devices* in Android 2.2 and 2.3. Unfortunately, the *pcsc* daemon and smartcard driver can only parse usb info from */proc/bus/usb* (Android 2.2 and 2.3; Mind that (1) CONFIG\_USB\_DEVICEFS=y. (2) Use *mount –t usbfs none /proc/bus/usb* to mount the usb file system.) and */dev/bus/usb* (Android 2.3 only; after doing some experiments, I found out that the */dev/bus/usb* node updates automatically even if the menuconfig is not set *CONFIG\_USB\_DEVICEFS=y*).

The *O\_DIRECT* attribute is not supported by Android.

*# modprobe w83627hf (dme1737)   
FATAL: Error inserting w83627hf (dme1737): Device or resource busy.*

That's a common problem with today’s ACPI hardware. Linux refuses to attach a driver to a device which is known to be also accessed by ACPI. Add "*acpi\_enforce\_resources=lax*" to the kernel command line to be able to load the *w83627hf* driver. In Android, add the command: #*BOARD\_KERNEL\_CMDLINE += apci\_enforce\_resources=lax* in *BoardConfig.mk*. You can use the command: *#cat /proc/cmdline* to check if the parameters are added into kernel command line successfully.

When Android x86 building system starts to build the kernel image, the x86 default *AndroidBoard.mk* is first read and then the *AndroidBoard.mk* which is located in device folder to invoked to check the prebuilt files. Unfortunately, if you add the command:*$(call add-prebuilt-targets,$(TARGET\_OUT),app/susidemo.apk)* in *AndroidBoard.mk*, the building system check these prebuilt modules/packages, which is time consuming, beforehand if you use # *make suidemo* to create *susidemo.apk*. Use *mm* or *mmm* command instead.

I install the some demos (*susidemo*, *smartcarddemo*…) as prebuilt packages so that I add *$(call add-prebuilt-targets,$(TARGET\_OUT),$(TARGET\_PREBUILT\_APPS))* and *$(call add-prebuilt-targets,$(TARGET\_OUT),$(TARGET\_PREBUILT\_LIBS))* in *AndroidBoard.mk*. The packages are put in *android/device/…* folder and will be copied to *android/out/…* folder while building kernel image.

If you create a package with JNI module inside (*susidemo* for the package name and *libsusijni.so* for the module name, for example), use the following command:# *make susidemo*. There are two output files including *susidemo.apk* which is copied into *android/out/…/system/app* and *libsusijni.so* which is copied into *android/out/…/symbol/system/lib*.

There are two ways to install the APK to target:

(1) Use # adb install susidemo.apk.

(2) copy susidemo.apk to *android/out/…/system/app* and *libsusijni.so* to *android/out/…/system/lib*. Not like Windows, it is not necessary for Android to install APK but copy the file to …/system/app. Mind that if this APK needs other SO due to jni, the relative SO is required to be copied into *…/system/lib*.

In JNI, if you want to get a string from a function, you can’t use *jstring* type, use *byte[]* instead. For example:

*byte[] byaData = new byte[128];*

*GetCPUName(byaData);*

*String strCPUName = new String(byaData); // Transform the array into a string in this way. Do not use toString() function.*

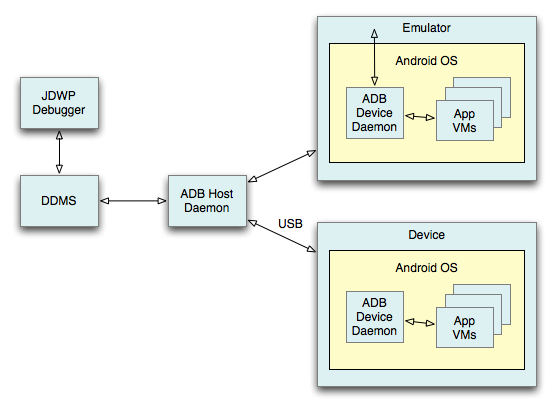
The Android APIs include a number of constants on the *android.os.Build* class that are intended to describe the current device. *android.os.Build.MODEL* represents a value chosen by the device implementer containing the name of the device as known to the end user. *android.os.Build.PRODUCT* represents a value chosen by the device implementer containing the development name or code name of the device.

SUSI library contains APIS to control watchdog. *OpenWdtDrv()* checks if */dev/watchdog* exists. *ioctl()* is invoked in *SusiWDTrigger()*, *SusiWDDisable()* and *SusiWDSetConfig()*.

R (R.java) cannot be resolved to a variable

To clarify, to fix this project to the right, you would have to change the name “Bubble.png” to “bubble.png”,   “Error.xml” to “error.xml”. You can rename “whatsUpMyGangstas.mp3″ to something like “whats\_up\_my\_gangsta5.mp3″.

For more detailed info, go to <http://www.mybringback.com/travis-android-help/1/hello-world/>



This picture shows how the various debugging tools work together in a typical debugging environment. For more info, go to <http://developer.android.com/tools/debugging/index.html>

# [Every Activity in Android is a Process,or One Application is one process](http://stackoverflow.com/questions/6468126/every-activity-in-android-is-a-process-or-one-application-is-one-process)

All activities inside an application run in one process?

It depends on value of android:process attribute in application manifest.

if attribute android:process is not defined for Application/Activity tags in the manifest, by default all the activities will run in single process (process name will be name of the package defined in manifest)

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.so.test" android:versionCode="1" android:versionName="1.0">

<application android:icon="@drawable/icon" android:label="@string/app\_name">

<activity android:name=".Activity1"></activity>

<activity android:name=".Activity2"></activity>

</application>

</manifest>

In the above manifest all activities run in process com.so.test, ps command output in adb shell:

# ps

app\_39 668 33 84492 20672 ffffffff afd0c51c S com.so.test

If android:process is specified for Activity new process will be created with the same userid and the activity runs in that process.

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.so.test" android:versionCode="1" android:versionName="1.0">

<application android:icon="@drawable/icon" android:label="@string/app\_name"

android:process="com.so.p1">

<activity android:name=".Activity1"></activity>

<activity android:name=".Activity2" android:process="com.so.p2"></activity>

</application>

</manifest>

If the manifest is defined like above

Activity1 runs in com.so.p1 process

Activity2 runs in com.so.p2 process

*ps* output in *adb shell*

# ps

app\_39 650 33 83192 20900 ffffffff afd0c51c S com.so.p1

app\_39 659 33 83188 20864 ffffffff afd0c51c S com.so.p2

If an Activity needs to be run in another process not defined in this manifest, both APKs should be signed with the same certificate.

So if the scenario is like this: The four Application components (Activity, Service, Content Provider, Broadcast Receiver) runs in one process, but they all share the main application thread.