深圳市九鼎创展技术有限公司技术文档

文档名称：《x210v3s android 平台用户手册》

深圳市九鼎创展科技有限公司

地址：深圳市宝安区兴业路宝安互联网产业基地 B 区 3003B 室

网址：http://www.9tripod.com

论坛：http://www.xboot.org

版权声明

敬告：

在售开发板的手册会经常更新，请在 http://www.9tripod.com 网站下载最新手册，不再另行通知。

版本说明

版本号	日期	作者	描述
Rev.01	2012-11-24	lqm	原始版本
Rev.02	2013-7-24	lqm	勘误

技术支持

一：论坛

在 BBS 论坛上发贴询问，是取得技术支持的有效途径，我司技术人员将在 24 小时内回贴，并尽可能的在 48 小时内解决所提出的问题。如遇节假日将顺延。

论坛网址：www.xboot.org

技术支持时间：周一到周五，9：00 到 18：00

二：邮件

在通过 BBS 论坛仍然无法解决的情况下，可以采用电子邮件的形式咨询。

邮箱地址：supports@9tripod.com

三：QQ 群

九鼎创展所有开发平台都对应有 QQ 交流群，QQ 群为广大客户提供一个共同交流讨论的地方，并不代表能够得到及时支持，技术人员并不能保证每时每刻在群里面盯着问题，很多技术问题并不能马上就能得到解决，请尽量在论坛上发贴提问，留给技术人员测试的时间，感谢您的支持。

网　　址：　www.9tripod.com

联系电话：　0755-61952306

E-mail：　supports@9tripod.com

销售与服务网络

公司：深圳市九鼎创展科技有限公司
地址：深圳市宝安区兴业路宝安互联网产业基地 B 区 3003B 室
邮编：518101
电话：4000033436　0755-33121205　0755-33133436
网址：http://www.9tripod.com
论坛：http://www.xboot.org
淘宝：http://armeasy.taobao.com
1688：http://armeasy.1688.com

QQ 群：
x4412/ibox4412一群：　【16073601】
x4412/ibox4412 二群：　【211126231】
x4418/ibox4418 论坛：　【199358213】
x6818/ibox6818 论坛：　【189920370】
x210/i210 一群：　【23831259】
x210/i210 二群：　【211127570】
x3288 技术论坛：　【159144256】

热烈欢迎广大同仁扫描右侧九鼎创展官方公众微信号，关注有礼，您将优先得知九鼎创展最新动态！

目录

第1章 android4.0 开发平台的搭建

Android 自从升级到 4.0 以来，相比之前的版本工程更加庞大，编译整套源码对 PC 机硬件要求很高，因此不建议采用虚拟机编译，强烈建议直接安装 Linux 操作系统，充分发挥 PC 机的性能。我们这里以 ubuntu12.04 64 位系统机器为例讲解，如果您是新手，建议与我们版本保持一致。

1.1 在 Windows_XP 系统下安装 ubuntu12.04

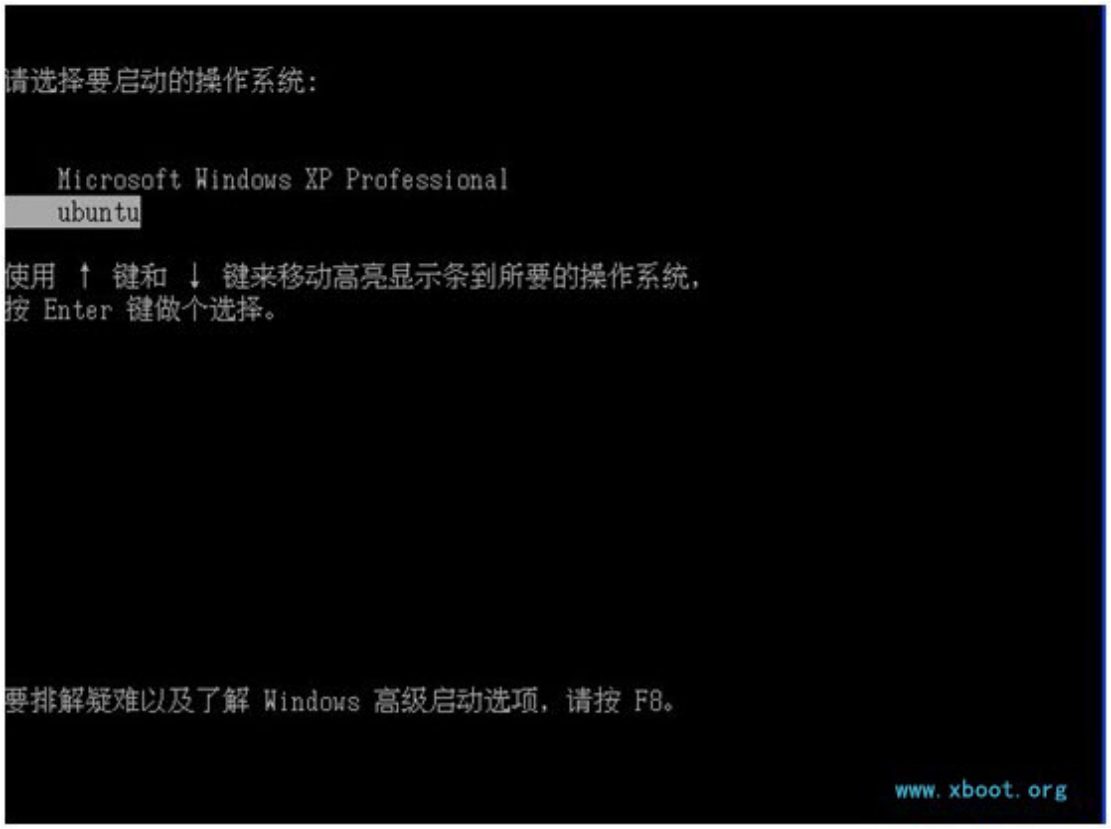
第一步：从光盘中获取 Grub4dos，或者网上下载 Grub4Dos，解压 grldr 和 menu.lst 两个文件至 XP 的 C 盘根目录下，然后修改 menu.lst 在末尾添加如下内容：

```
title Install Ubuntu
root (hd0,0)
kernel (hd0,0)/vmlinuz boot=casper iso-scan/filename=/Ubuntu-10.10-desktop-i386.iso ro quiet splash
locale=zh_CN.UTF-8
initrd (hd0,0)/initrd.lz
```

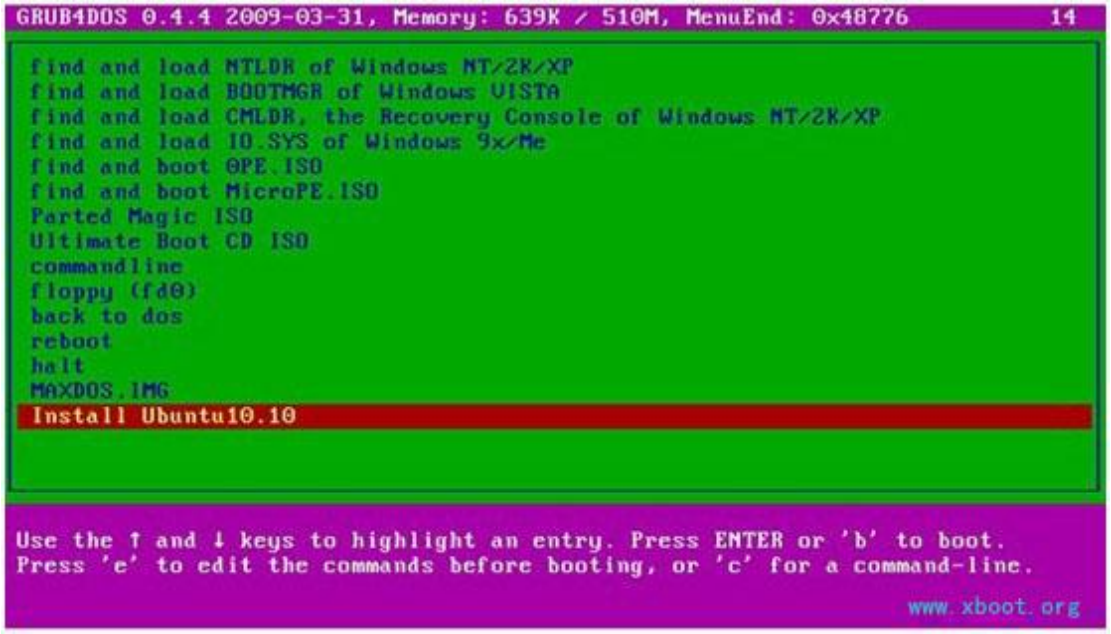
第二步：修改 Windows XP 的 boot.ini 文件，在命令提示符下去掉 boot.ini 的相关属性：attrib -s -h -r c:\boot.ini，然后编辑 boot.ini 在末尾添加：C:\grldr="ubuntu"

第三步：将下载好的 ubuntu 镜像文件直接放在 C 盘根目录，将 ubuntu 镜像中 casper 目录下的 initrd.lz 和 vmlinuz 两个文件也解压至 C 盘根目录

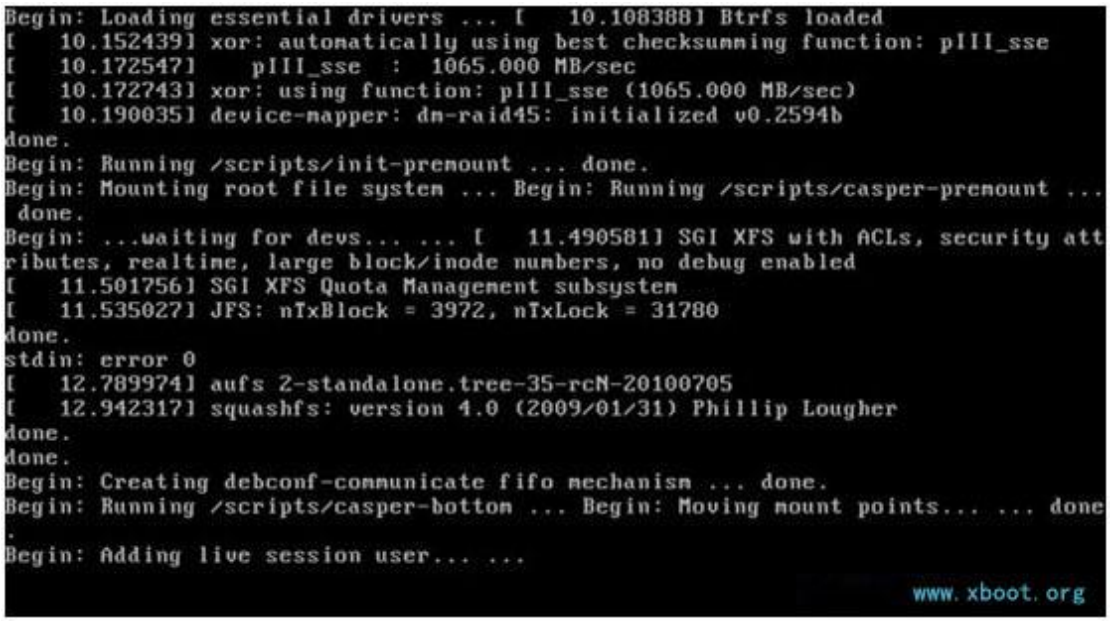
第四步：重启系统，就可以进行安装了，在启动菜单中，已经有 ubuntu 的安装选项，如下图所示：



选择 ubuntu，进入如下界面：



选择最后一项，Install ubuntu10.10，开始从 C 盘读取 ISO 文件：



紧接着会进入 Ubuntu010.10 live CD 的安装界面：

第五步：在安装过程的分区步骤中会有不能卸载/isodevice 的提示。所以在安装前请在命令窗口终端输入：

```
sudo umount  -l  /isodevice
```

第六步：双击安装 ubuntu10.10 的图标，开始安装：

选择要安装的语言，点击前进按钮：

这里可以不勾选安装中下载更新的选项，可以在后期安装完成后更新，当然也可以勾选，在安装过程中更新。后面的安装这个第三方软件的选项，推荐勾选，用于播放 mp3。点击前进：

注意，这了与 windows 系统并存，这里一定要选择手动指定分区，点击前进：

注意，安装 Linux 需要在硬盘中建立 Linux 分区，可以把系统文件分几个区来装（必须说明载入点），也可以只装在一个分区中（载入点是“/”），通常情况下至少应该创建以下几个分区：

Swap 分区：交换分区 Swap 的功能和 Windows 下的交换文件相同，都是作为虚拟内存使用，其大小一般设置为内存的两倍大小（内存少于 256MB 时）或和内存一样（内存为 256MB 及以上时）。

/boot 分区：/boot 分区用于引导系统，它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是有必要的，如果有了一个单独的/boot 启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在 50MB～100MB 之间。

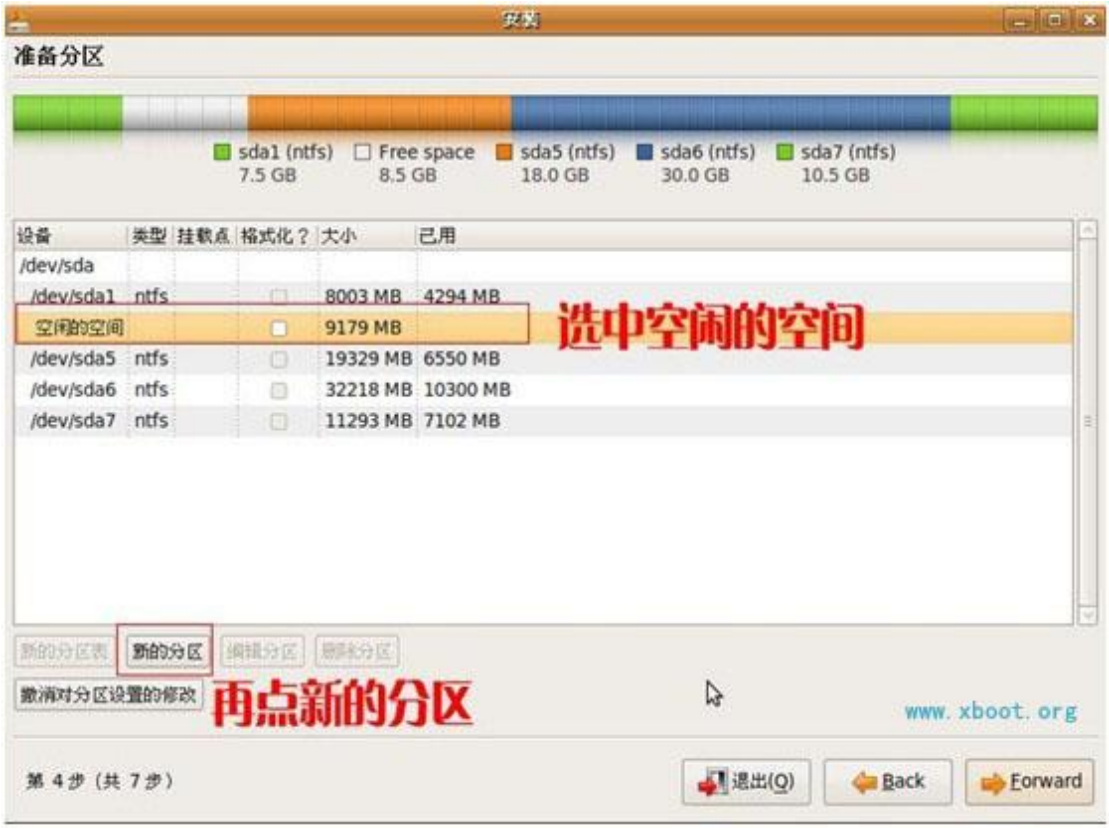
/（根）分区：Linux 的大部分系统文件和用户文件都保存在/（根）分区上，所以该分区一定要足够大。比如 Red Hat Linux 完全安装一般大小在 5G 左右，所以该分区大小一般大于 5GB。

紧接着前面的步骤，点击前进按钮：

选择要安装的分区盘，点击编辑分区：

填入要改变的分区的大小，点击 OK：

弹出确认修改的对话框，点击继续，



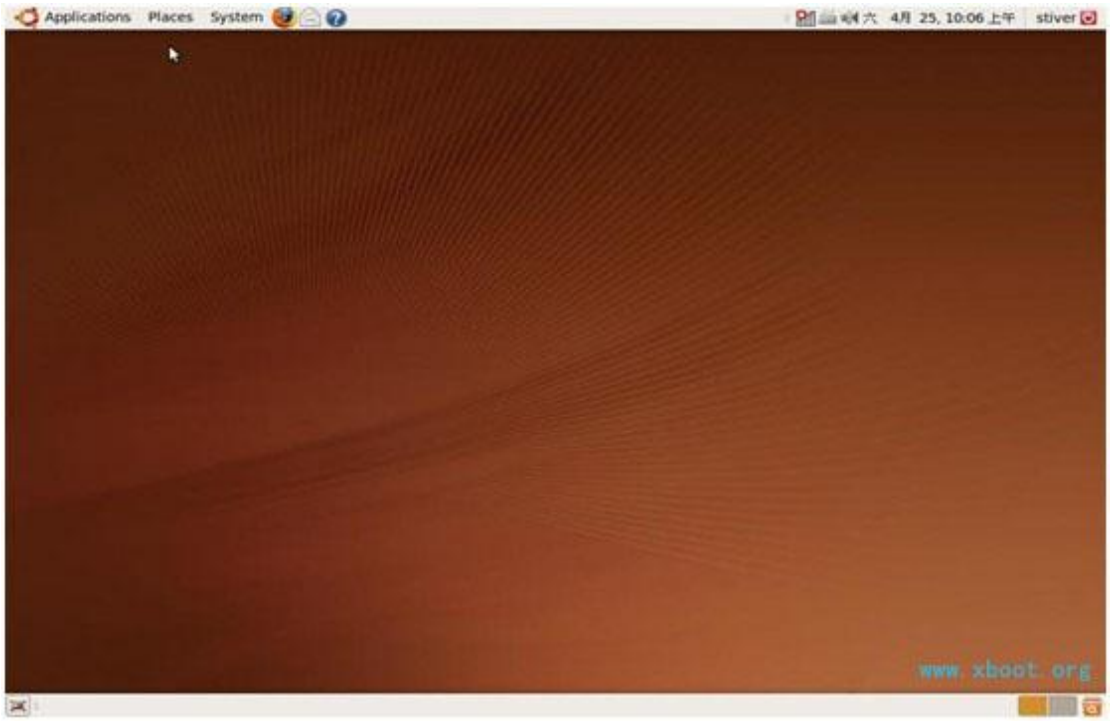
这时已经从 C 盘分出了 8G 的空间，再点击新的分区，

新建用于虚拟内存的交换空间分区，分区容量设置为 512，也可以设置为 1024，点击确定，

用同样的步骤新建 ext3 的分区，点击确定，再点击前进按钮，

填入姓名，计算机名，用户名，密码等，点击前进按钮：

重启后首次进入 ubuntu 的桌面，说明系统已安装成功，如下图：



1.2 使用 U 盘安装 ubuntu

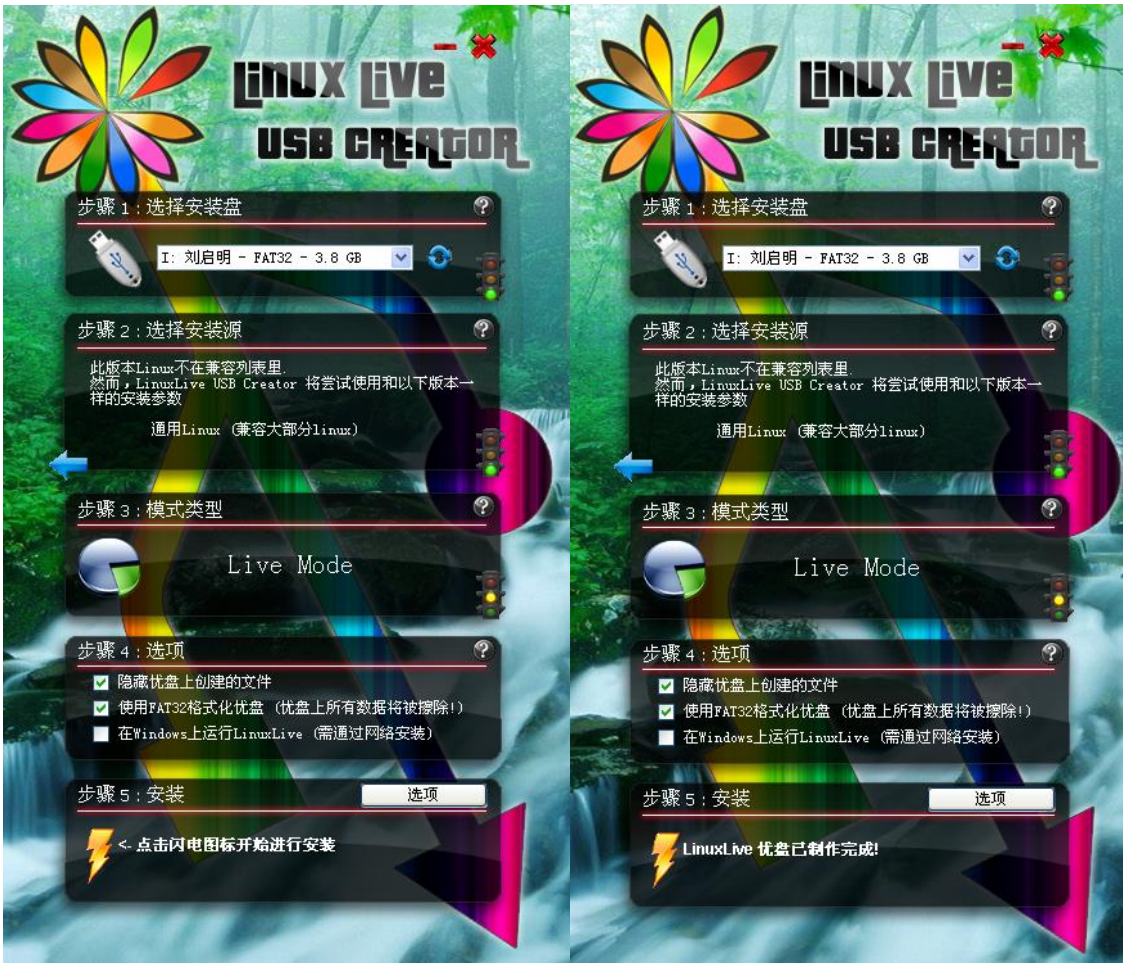
安装工具：

- 2G 以上 U 盘一个
- lili usb creater 软件，下载地址：http://www.linuxliveusb.com/
- ubuntu 最新系统，下载地址：http://cdimages.ubuntu.com/
- PC 机一台

安装方法：

第一步：下载好 ubuntu 的 ISO 文件，和 lili usb creater 这个软件并安装。

第二步：插入 usb，并打开 usb creater 这个软件，根据软件提示设置，在步骤 1 中选择安装盘，找到识别出的 U 盘；在步骤 2 中找到下载的 ubuntu 映像文件；步骤 3 默认，步骤 4 中选中隐藏优盘上创建的文件，使用 FAT32 格式化 U 盘；最后在步骤 5 中点击闪电图标开始安装，直到提示优盘已安装完成为止。



第三步：重启电脑，开机时，看清屏幕下方的提示，进入 BIOS 设置菜单，选择 U 盘启动。一般台式机是按 DEL 键，笔记本有些是按 F2，有些按 F10 进入。设置完成后保存退出。

第四步：再次重启系统，这时已经可以看到 ubuntu 的安装界面了，选择中文，继续；

第五步：选择 install （您也可以选择 live mode 可以体验下系统），继续；

第六步：也选择中文，点击继续：再继续；

第七步：配置网络，可以安装时升级，也可以不升级，等安装完系统后再手动升级；

第八步：第一个选项是把以前的系统升级到 ubuntu11.10，如果你只要单系统的可以选择第一项，第二项是升级到 ubuntu11.10 并把其他的资料删除掉，第三项是我们用的最多的，比较灵活，选 something else，继续，在这里我分出了两个区给 ubuntu，一个 / 和 一个 /home ，分区可以新建，可以对它格式化，具体根据需要选择；

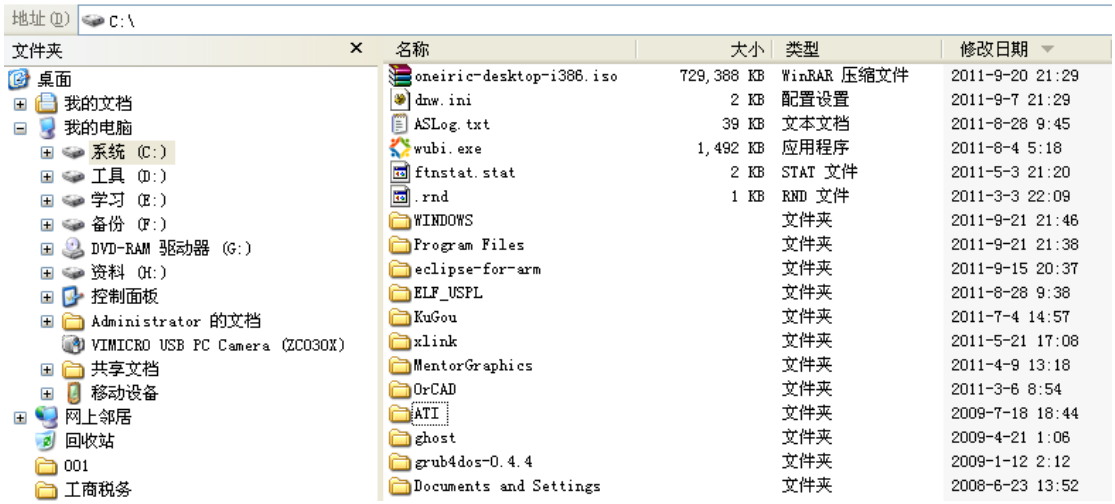
第九步：这是设置区域，选择上海就行了；

第十步：选择键盘布局。选中国；

第十一步：选择用户名和密码，到此配置完毕，点击继续直接安装，喝一会儿咖啡，待安装完毕，重启之后，就可以看到美丽的 ubuntu 世界了。

1.3 使用 wubi 安装 ubuntu 系统

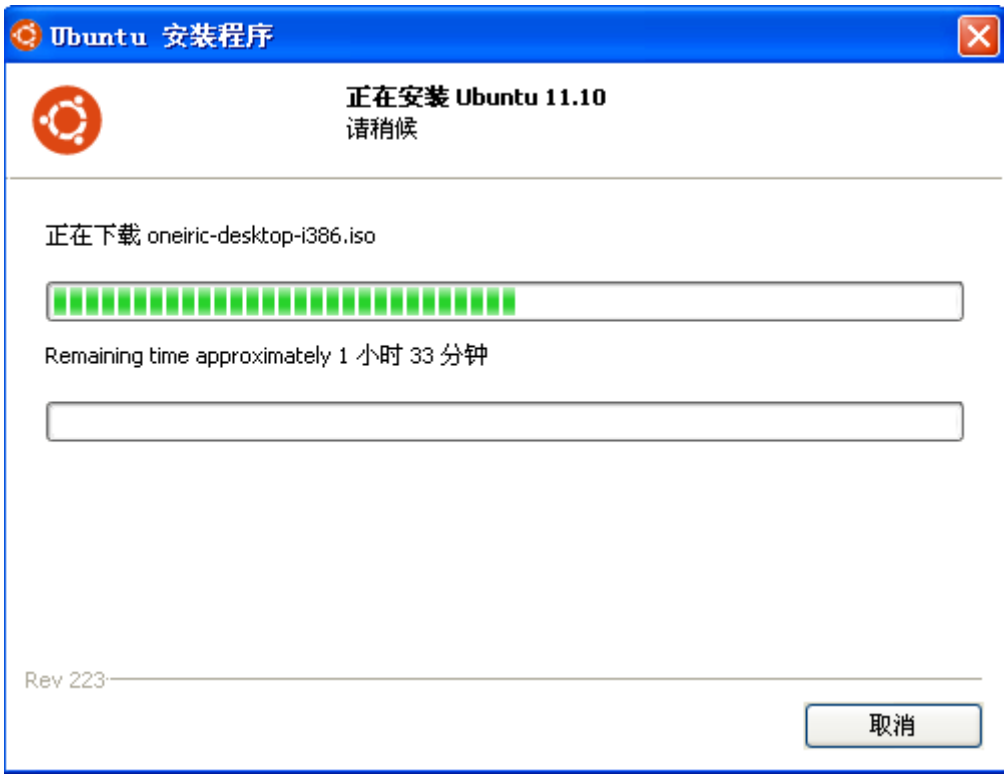
第一步：用解压软件把 ubuntu 的 iso 文件，提取出 wubi 软件，把这两个放在某个分区的更目录下：



第二步： 双击 wubi：

选择目标驱动器以及安装大小，以下按图示安装即可：


Ubuntu 安装程序
您正要安装 Ubuntu 11.10
请为新帐户选择用户名称和密码
目标驱动器：
H: (44GB 空闲)
语言：
Chinese (CN)
安装大小：
30GB
用户名：
administrator
口令：
桌面环境：
Ubuntu
请输入有效密码。
Rev 223
辅助功能
安装
取消


Ubuntu 安装程序
正在安装 Ubuntu 11.10
请稍候
正在下载 oneiric-desktop-i386.iso
Remaining time approximately 1 小时 33 分钟
Rev 223
取消

1.4 设置 XP 为开机默认启动

ubuntu 安装后每次开机都是默认进入 ubuntu 系统的，对于以 windows 为主的朋友，每次开机都要守在画面切换到 XP 启动，可见十分麻烦，通过下面，你可以设置让你的 XP 系统为第一启动。

修改/boot/grub/grub.cfg 文件，可以看到最末尾有如下语句：

```
menuentry "Microsoft Windows XP Professional (on /dev/sda1)" {
        insmod part_msdos
        insmod ntfs
        set root='(hd0,msdos1)'
        search --no-floppy --fs-uuid --set 0E48A65048A6367D
        drivemap -s (hd0) ${root}
        chainloader +1
}
### END /etc/grub.d/30_os-prober ###
```

仔细阅读该文件，发现有不少 menuentry，这些正是对应了开机启动时的选择项，将上面语句放在第一个 menuentry 的前面，这样 Windows XP 就为默认的启动系统了。修改后的部分代码如下：

```
### BEGIN /etc/grub.d/05_debian_theme ###
set menu_color_normal=white/black
set menu_color_highlight=black/light-gray
### END /etc/grub.d/05_debian_theme ###

### BEGIN /etc/grub.d/30_os-prober ###
```

```
menuentry "Microsoft Windows XP Professional (on /dev/sda1)" {
        insmod part_msdos
        insmod ntfs
        set root='(hd0,msdos1)'
        search --no-floppy --fs-uuid --set 0E48A65048A6367D
        drivemap -s (hd0) ${root}
        chainloader +1
}
### END /etc/grub.d/30_os-prober ###

### BEGIN /etc/grub.d/10_linux ###
menuentry 'Ubuntu, with Linux 2.6.35-30-generic' --class ubuntu --class gnu-linux --class gnu
--class os {
        recordfail
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos9)'
        search --no-floppy --fs-uuid --set 4833f619-6388-4dd3-acd7-6fa3eacb9a15
        linux       /boot/vmlinuz-2.6.35-30-generic
root=UUID=4833f619-6388-4dd3-acd7-6fa3eacb9a15 ro      quiet splash
        initrd      /boot/initrd.img-2.6.35-30-generic
}
```

1.5 ubuntu 下磁盘格式化

在做开发时，经常会把 SD 卡格式化为 msdos，ext3 等格式。这时，放在 windows 下将无法格式化，只能借助于 Linux。

在 Linux 下使用 fdisk 和 mkfs 两个工具实现 SD 卡的格式化。首先，使用 fdisk 指令删除里面的所有分区，步骤为：

```
fdisk   /dev/sdb
d
w
```

每输一次 d，输入一次分区的序号，直到删完，再按 w 写入，完成分区的删除。再按 n,回车，回车，新建一个分区。之后再使用 mkfs 工具格式化为指定格式的盘，如格式化为 fat32 格式，则执行如下指令：

```
mkfs   –t   vfat   /dev/sdb
```

1.6 Ubuntu 下通过 SSH 远程登录服务器

第一步：在服务器上安装 ssh 的服务器端。

```
apt-get install openssh-server
```

第二步：启动 ssh-server。

```
service ssh restart
```

第三步：确认 ssh-server 已经正常工作。

```
netstat -tlp
```

有如下打印信息：

```
tcp 0 0 *:ssh *:* LISTEN -
```

看到上面这一行输出说明 ssh-server 已经在运行了。

第四步：在 Ubuntu 客户端通过 ssh 登录服务器。假设服务器的 IP 地址是 172.18.0.198，登录的用户名是 liuqiming。

```
$ ssh -l liuqiming 172.18.0.198
```

接下来会提示输入密码，然后就能成功登录到服务器上了。可以通过资源管理器浏览的方式登录服务器，点击位置->连接到服务器，服务器类型选择 ssh，服务器一栏填入服务器的 IP 地址，点击连接即可。

1.7 ubuntu 下使用邮箱

ubuntu 下默认就有邮箱软件，如在使用时发现无法接受或发送邮件，在发送电子邮件->身份验证点击检查的类型，再选择没有划斜线的就好了。

1.8 ubuntu 下安装五笔

使用如下指令：

```
sudo apt-get install ibus-tables-wubi
```

然后在系统->首选项->键盘输入法的输入法中选择 WUBI，添加进去即可。

1.9 ubuntu 下安装 chrome 浏览器

ubuntu10.10 默认安装的火狐浏览器，如果用户喜欢该浏览器，可以跳过此节。google 出了基于 linux 的浏览器 chrome，用户可以使用如下方式安装。在网上下载安装源文件，http://tools.google.com/chrome/，也可以从光盘中获得，双击即可安装。

1.10 ubuntu 下安装 VIM

使用如下命令安装即可：

```
sudo apt-get install vim
```

1.11 ubuntu 打开 WINDOWS 下记事本乱码问题

出现这种情况的原因是，gedit 使用一个编码匹配列表，只有在这个列表中的编码才会进行匹配，不在这个列表中的编码将显示为乱码。您要做的就是将 GB18030 加入这个匹配列表。

您可以遵循以下步骤，使您的 gedit 正确显示中文编码文件。

1. 终端中键入“gconf-editor”，并按下回车键，打开“配置编辑器”。
2. 展开左边的树节点，找到 /apps/gedit-2/preferences/encodings 节点并单击它。
3. 双击右边的 auto_detected 键，打开“编辑键”对话框。
4. 单击列表右边的“添加”按钮，输入“GB18030”，单击确定按钮。
5. 列表的最底部新增加了一个“GB18030”。单击右边的向上，将“GB18030”放在第二位；
6. 单击确定按钮，关闭配置编辑器。

现在，gedit 应该能够顺利打开 GB18030 编码的文本文件了。如果不放心，可以再增加 GBK、GB2312 编码。

1.12 ubuntu 下安装源码比较工具

ubuntu 下源码比较工具很多，比较常见的是 meld 工具，使用如下指令安装：

```
sudo apt-get install meld
```

安装完成后，可以在应用程序->编程中打开。

1.13 ubuntu 下安装串口终端 minicom

使用如下指令安装：

```
sudo apt-get install minicom
```

安装完成后，需要设置 minicom。如果直接使用串口，通常设置为 ttyS0，如果使用 USB 转串口，通常设置为 ttyUSB0。输入如下指令：

```
sudo minicom -s
```

选择 Serial port setup，选择 A，输入正确的串口终端，选择 E，输入 115200 8N1，选择 F 和 G，都设置为 No，不使用流控，再回车，选择 Save setup as dfl。注意，只有 root 用户才有权限保存参数。笔记本用户通常使用的 USB 转串口延长线，目前市面上大多都是 pl2303 方案，插上 USB 转串口延长线后，输入如下命令查询驱动是否正常加载：

```
lsmod |grep pl2303
```

正常加载时会提示如下信息：

```
lqm@lqm:~$ lsmod |grep pl2303
pl2303                  11756  1
usbserial               33100  3 pl2303
```

再使用如下命令查询系统的一些信息：

```
dmesg | tail -f
```

正常情况下会出现如下提示：

```
lqm@lqm:~$ dmesg |tail -f
[  383.093851] ERROR! H2M_MAILBOX still hold by MCU. command fail
[  383.148849] ---> RTMPFreeTxRxRingMemory
[  383.148903] <--- RTMPFreeTxRxRingMemory
[  383.180580]  RTUSB disconnect successfully
[  387.762330] usb 2-4: USB disconnect, address 3
[  387.762566] pl2303 ttyUSB0: pl2303 converter now disconnected from ttyUSB0
[  387.762601] pl2303 2-4:1.0: device disconnected
[  392.164589] usb 2-4: new full speed USB device using ohci_hcd and address 5
[  392.379898] pl2303 2-4:1.0: pl2303 converter detected
[  392.412998] usb 2-4: pl2303 converter now attached to ttyUSB0
```

表示串口设备名称为 ttyUSB0。有时会提示如下错误：

```
lqm@lqm:~$ dmesg | tail -f
[  408.910351] 0x1300 = 00073200
[  413.945752] ===>rt_ioctl_giwscan. 8(8) BSS returned, data->length = 1177
[  419.047006] ===>rt_ioctl_giwscan. 7(7) BSS returned, data->length = 1067
[  419.047302] ==>rt_ioctl_siwfreq::SIOCSIWFREQ[cmd=0x8b04] (Channel=1)
[  419.392535] wlan0: no IPv6 routers present
```

```
[   433.902136] ===>rt_ioctl_giwscan. 6(6) BSS returned, data->length = 960
[   473.902907] ===>rt_ioctl_giwscan. 8(8) BSS returned, data->length = 1271
[   533.900777] ===>rt_ioctl_giwscan. 8(8) BSS returned, data->length = 1200
[   613.904091] ===>rt_ioctl_giwscan. 9(9) BSS returned, data->length = 1435
[   713.904199] ===>rt_ioctl_giwscan. 8(8) BSS returned, data->length = 1263
```

一般情况下重插拔一次 USB 转串口线即可。

1.14 ubuntu 卡死的解决办法

ubuntu 系统有时也会像 windows 系统一样，卡死不动。这时除了复位系统，我们也可以尝试如下方法：

一：按住 ctrl+alt+F2 进入 tty2；

二：查看进程：

```
ps -e
```

三：kill 掉相关进程

四：再按住 alt+F7 返回图形界面

第2章 android 开发工具

2.1 代码编辑工具

在 windows 下开发时，很多人都习惯使用 source insight，但是 source insight 并没有 linux 版本，而且自从 3.5 版本之后，就再也没有更新了。

在 linux 下，同样也有很多优秀的代码编辑软件，如 Emacs、KVIM、Arachnophilia、Bluefish、Komodo Edit、NEdit、Gedit、Kate、Quanta Plus 等等。这里介绍两款比较优秀的代码编辑软件，slickedit 和 eclipse。

2.1.1 slickedit

安装步骤如下：

第一步：解压 se_14000202_linux_full.tar.gz：

```
tar -zxvf se_14000202_linux_full.tar.gz
```

第二步：进入解压的目录，运行安装程序 vsinst。这里要加上 sudo，增加读写访问权限：

```
sudo ./vsinst
```

第三步：会弹出安装信息，按住回车不放，直到弹出如下提示：

```
Do you agree to the above license terms?[yes or no]
```

输入 yes

第四步：弹出如下信息：

```
Install directory [/opt/slickedit]:
```

这里提示输入安装路径，默认按回车即可

第五步：提示如下信息：

```
Directory /opt/slickedit/ does not exist. Create [Y]?
```

输入 Y，回车，程序开始安装。

第六步：安装过程中会弹出一个 SlickEdit License Manager 的对话框，点退出

再弹出一个对话框，点 OK

第七步：这时会提示：

```
INSTALLATION SUCCESSFULLY COMPLETED
1.Type "/opt/slickedit/bin/vs" to run SlickEdit.
2.You may want to add "/opt/slickedit/bin/" to your users' PATH.
```

第八步：退回原存放安装文件的目录，解压破解文件

```
slickedit2009-14.0.2.2-linux-cracked.tar.gz：
tar -xvf slickedit2009-14.0.2.2-linux-cracked.tar.gz
```

将解压出的破解文件 VS 拷备到/opt/slickedit/bin 目录下：

```
sudo cp vs /opt/slickedit/bin
```

第九步：在/opt/slickedit/bin 目录下，执行./vs 命令打开 slickedit 软件，看看是否大功告成？以下是执行命令：

```
cd /
./opt/slickedit/bin/vs
```

第十步：启动方式

可以在 [终端] 中 ./vs 启动 ，也可以自己创建一个起动器。

```
cd /opt/slickedit/bitmaps
```

找到图标文件，slickedit 2010 下我选择了 vse_profile_256.bmp

```
$ sudo cp vse_profile_256.bmp /usr/share/icons
```

一般把图标都放在/usr/share/icons 下

```
$ cd /usr/share/applications
$ sudo gedit slickedit.desktop
```

输入如下语句：

```
[Desktop Entry]
Name=Slickedit
Comment=Slickedit
Exec=/opt/slickedit/bin/vs
Icon=/usr/share/icons/vse_profile_256.bmp
Terminal=false
Type=Application
Categories=Development;
StartupNotify=true
```

这时，在应用程序->编程中，就能找到 slickedit 的图标了。

第十一步：如果你实在是用烦了这个软件，那就干掉他吧！进入/opt 目录，输入如下指令：

```
rm -rf slickedit
```

从此让他滚得越远越好！

附：slickedit 行号显示：

```
tool->options->Languages->Application Languages->C/C++->View->Line numbers
```

2.1.2 eclipse

1. 在 ubuntu 下安装 eclipse

第一步：进入如下网站下载 eclipse:

http://www.eclipse.org/

选择 Eclipse IDE for Java Developers,Linux 32 Bit 或 Linux 64 Bit 根据自己的机器而定；

第二步：将下载的文件解压到用户目录：

```
cp eclipse-java-indigo-SR1-linux-gtk.tar.gz ~/
cd ~
tar xf eclipse-java-indigo-SR1-linux-gtk.tar.gz
```

第三步：创建启动图标：

```
sudo cp icon.xpm /usr/share/icons/eclipse.xpm
sudo gedit /usr/share/applications/eclipse.desktop
```

输入如下语句：

```
[Desktop Entry]
Name=eclipse
Comment=eclipse
Exec=/home/lqm/eclipse/eclipse
```

```
Icon=/usr/share/icons/eclipse.xpm
Terminal=false
Type=Application
Categories=Development;
StartupNotify=true
```

这时，在应用程序->编程中，就能找到 eclipse 图标了，点击即可启动 eclipse.

第四步：安装 CDT 插件

安装 eclipse 后，还不能建立 C/C++工程，需安装插件。进入官网下载：

http://www.eclipse.org/downloads/download.php?file=/tools/cdt

解压下载的文件 cdt-master-8.0.1.zip，将解压出来的 plugins 和 features 目录拷贝到 eclipse 安装目录，直接合并即可完成安装。

2. 使用 eclipse 新建一个工程

第一步：打开 eclipse，首次打开时，会提示选择工作路径，建立自己的路径，确定即可；

第二步：新建一个工程。点击 File->New->Project，选择 C/C++->C Project，点击 Next，在 Project name 一栏输入工程名称，如 xboot，在 Project name 下面有一个 Use default location 的选择框，去掉前面的勾，点击 Browse，指向我们需要修改的文件的目录。

在 Project type 中选择一个工程类型，如 Shared Library->Empty Project，在 Toolchains 中选择 Cross GCC，再点击 Next，在 Select Configurations 中选择配置类型，如 Release，点击 Finish 完成。

第三步：这时在 Project Explorer 中有 xboot 的目录，右击 xboot，点击 Import，找到 General->File System，双击，弹出 Import 对话框，在 From directory 中找到需要加载的文件的目录，点击 Select All，将把加载的目录的所有类型文件添加到工程中，点击 Finish 按钮，提示是否覆盖.cproject，点击 Yes To All，这时我们需要编辑的文件就已经全加载到工程中了。

2.2 adb 工具

2.2.1 安装 adb 工具

网上下载最新的 SDK，下载地址为：

http://developer.android.com/sdk/index.html

对于 WINDOWS 系统，需下载 installer_r12-windows.exe，如下图所示：

Platform	Package	Size	MD5 Checksum
Windows	android-sdk_r12-windows.zip	36486190 bytes	8d6c104a34cd2577c5506c55d981aebf
	installer_r12-windows.exe (Recommended)	36531492 bytes	367f0ed4ecd70aefc290d1f7dcb578ab
Mac OS X (intel)	android-sdk_r12-mac_x86.zip	30231118 bytes	341544e4572b4b1afab123ab817086e7
Linux (i386)	android-sdk_r12-linux_x86.tgz	30034243 bytes	f8485275a8dee3d1929936ed538ee99a

如果 PC 机上没有安装 JDK，会提示需要先安装，需从 java 官网下载，如 jdk-6u25-windows-i586.exe，不同的版本名称不一样。安装完 JDK 后，再安装 SDK，默认会装在 C 盘，建议安装在 D 盘，这时 adb 工具在下面的路径：

D:\Program Files\Android\android-sdk\platform-tools

修改系统环境变量，找到 Path 环境变量，在前面添加

D:\Program Files\Android\android-sdk\platform-tools;

注意一定要加一个分号隔开。然后在 WINDOWS 的 CMD 命令行中输入 adb，将会弹出 adb 的一些参数。如果出现”adb 不是内部或外部命令的错误”，表示系统没有找到 adb，将上面目录中的 adb.exe 和 AdbWinApi.dll 拷贝到 C:\WINDOWS\system32 中即可。

2.2.2 查看设备的连接状态

启动 x210 开发板，进入控制台，然后用 USB 线连通开发板与 PC 机，首次连接时，会提示需要安装驱动，点手动安装，指向如下路径：

```
D:\Program Files\Android\android-sdk\extras\google\usb_driver
```

或者指向驱动程序 x210_android_driver 的路径，点下一步安装直到完成。

进入 WINDOWS 下的 CMD 命令行，输入如下命令验证开发板是否连接：

```
adb  devices
```

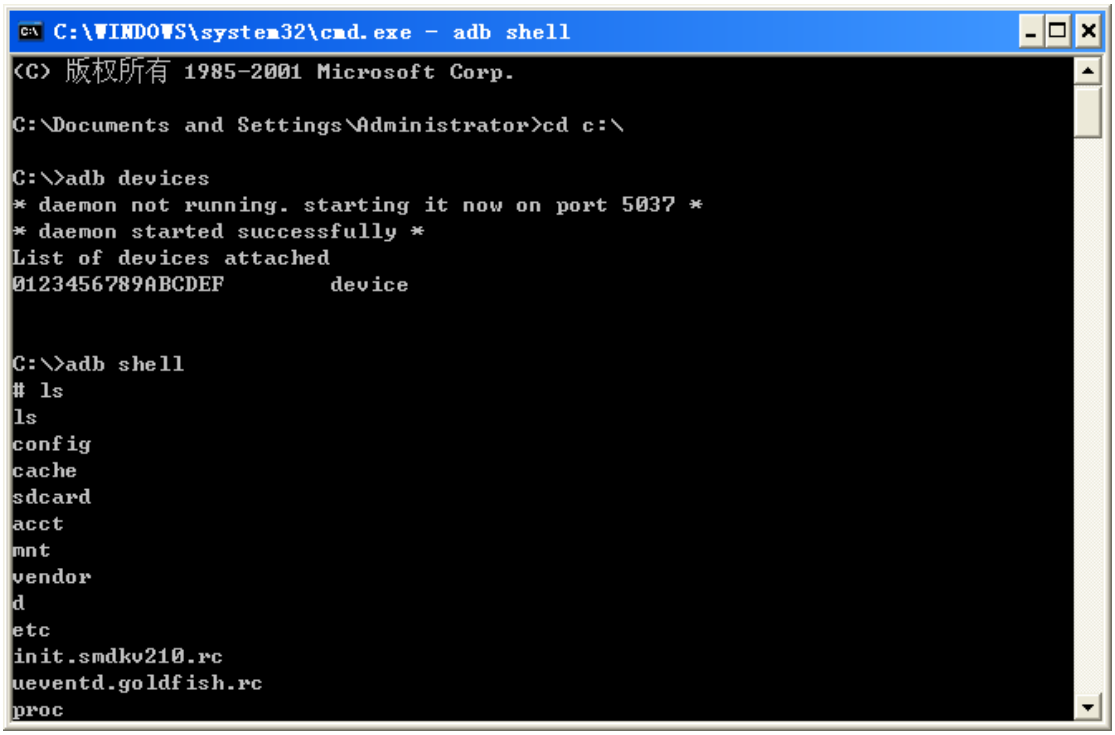
显示下面内容表示成功连接：

2.2.3 进入 adb shell

使用如下命令进入开发板终端：

```
adb  shell
```

如下图所示：



输入 exit 退回 DOS 操作界面。注意，有时候，执行 adb devices 命令时，会提示 error: more than one device and emulator，很有可能是播放了 USB 设备造成的。这时已经无法再通过 adb 传输数据，解决的办法很简单，如果是使用 windows，直接在进程中干掉 adb.exe，再启动 adb 即可。

第3章 android4.0 源码开发环境的搭建

3.1 安装 android 源码依赖包

说明：本文档所有开发全部基于 ubuntu12.04 64 位系统，后续不再声明。

依赖的软件包：

- GIT
- JDK 6.0
- flex，bison，gperf，libsdl-dev，libesd0-dev，libwxgtk2.6-dev，build-essential，zip，curl，genromfs

使用如下命令安装所需的软件包：

```
sudo apt-get install git-core gnupg sun-java6-jdk flex bison gperf libsdl-dev libwxgtk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev genromfs
```

很可能个别软件包会安装失败，比如 sun-java6-jdk，这时需要我们手动来安装。下面是在 ubuntu 下手动安装 jdk1.6 的详细步骤：

第一步：在官网下载最新的 jdk1.6 的安装源文件 jdk-6u27-linux-i586.bin，下载地址为：http://www.oracle.com/technetwork/java/javase/downloads，也可以从光盘中获得。

第二步：将下载的文件复制到/usr/lib/jvm 目录，执行如下命令安装：

```
chmod +x jdk-6u27-linux-i586.bin
sudo ./jdk-6u27-linux-i586.bin
```

第三步：修改环境变量：

```
sudo gedit /etc/profile
```

在最末尾加入如下语句：

```
#set java environment
JAVA_HOME=/usr/lib/jvm/jdk1.6.0_27
export JRE_HOME=/usr/lib/jvm/jdk1.6.0_27/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

第四步：重启系统

第五步：查看当前 java 安装版本：

```
root@lqm:/usr/lib/jvm# java -version
java version "1.6.0_27"
Java(TM) SE Runtime Environment (build 1.6.0_27-b07)
Java HotSpot(TM) Server VM (build 20.2-b06, mixed mode)
root@lqm:/usr/lib/jvm#
```

到此，jdk1.6 成功安装。

注意，如果安装了 jdk1.5 和 1.6,很有可能查询版本时，仍然是 1.5 的，执行如下指令：

```
sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk1.6.0_27/bin/java 255
sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk1.6.0_27/bin/javac 255
```

这两条指令用于创建符号链接。其中，/usr/bin/java 是不用改动的，为你的原有的 jdk 路径，/usr/lib/jvm/jdk1.6.0_27/bin/java 这个是 jdk1.6 的 java 路径 255 是优先级。

再执行如下命令：

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
```

弹出如下对话框：

```
root@lqm:/usr/local# update-alternatives --config java
There are 2 choices for the alternative java (providing /usr/bin/java).
  选择         路径                                            优先级  状态
------------------------------------------------------------
* 0            /usr/lib/jvm/jdk1.6.0_27/bin/java               255        自动模式
  1            /usr/lib/jvm/java-1.5.0-sun/jre/bin/java        53         手动模式
  2            /usr/lib/jvm/jdk1.6.0_27/bin/java               255        手动模式
```

要维持当前值[*]请按回车键，或者键入选择的编号。这里选择我们需要使用的 jdk 版本，回车即可。

说明：jdk1.6 也可以使用如下方法安装：

手动修改下载源，指令如下：

```
cd /etc/apt
cp sources.list sources.list.bak
vim sources.list
```

在最末行添加如下语句：

```
deb http://archive.canonical.com/ubuntu maverick partner
```

然后更新源：

```
apt-get update
```

再安装 java6:

```
apt-get install sun-java6-jdk
```

3.2 安装交叉编译工具链

将光盘中 toolchain 目录下的交叉编译工具拷贝到用户名目录下，并解压：

```
cp   yourcdromdir/toolchain/*   ~/          #复制交叉编译工具链
sudo  tar  xvf  arm-2009q3.tar.bz2  -C  /usr/local/arm
```

至此，交叉编译工具链安装成功。

3.3 安装 64 位系统必要的一些补丁包

```
apt-get install lsb-corelibc6-dev-i386g++-multiliblib32z1-devlib32ncurses5-dev
```

3.4 指定 GCC 交叉编译器

在 ubuntu 系统上安装最新的 GCC 交叉编译器时，版本已经超过 4.4 了，使用如下指令可查询 GCC 的版本：

```
gcc –version
```

可能出现的界面如下：

```
terry@ubuntu-server:~$ gcc --version
gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Copyright © 2011 Free Software Foundation, Inc.
terry@ubuntu-server:~$
```

上面显示 4.6.3 版本，默认我们给出的包，在 4.6.3 上编译会提示一些错误，都是新的 GCC 引出的错误，网上都有解决办法，如果不想修改这些错误，可将 GCC 版本降至 4.4 即可。解决办法 ：

```
sudo apt-get install gcc-4.4 g++-4.4 g++-4.4-multilib
```

装完后，开始降级 gcc ,这不会影响系统，仅仅是改个链接而已，不喜欢的话改回来即可。

```
cd /usr/bin
sudo mv gcc gcc.bk
sudo ln -s gcc-4.4 gcc
sudo mv g++ g++.bk
sudo ln -s g++-4.4 g++
```

3.5 安装 android 源码包

从光盘中拷贝 android 源码包 x210_ics_rtm_v11.tar.bz2，放在自己的用户名目录，注意最好不要放在文件系统的根目录，这样会出现管理权限问题。

示例方法：在用户权限下执行如下命令：

```
cp    yourcdromdir/android4.0.4/bsp/x210_ics_rtm_v11.tar.bz2 ~/
tar xvf x210_ics_rtm_v11.tar.bz2
```

这时，整个 android 文件系统全部都放在了当前解压的目录中。至此，android 源码包安装完成。

说明：源码包名称可能会因发布日期等有所不同，具体以光盘中实际名称为准。

第4章 android 脚本分析

4.1 源码编译脚本

编译脚本 mk 内容及注释如下：

```
#!/bin/sh
#
# Description       : Build Android Script.
# Authors       : http://www.9tripod.com
# Version       : 0.01
# Notes             : None
#
# 环境声明
export ANDROID_JAVA_HOME=/usr/lib/jvm/java-6-sun/
SOURCE_DIR=$(cd `dirname $0` ; pwd) #找到脚本所在路径，即 android 源码根目录

TOOLS_DIR=${SOURCE_DIR}/tools       # 必要的一些工具路径
RELEASE_DIR=${SOURCE_DIR}/out/release  # 生成映像汇总的路径
TARGET_DIR=${SOURCE_DIR}/out/target/product/x210       # 编译后生成的路径
KERNEL_DIR=${SOURCE_DIR}/kernel   #内核路径
BOOTLOADER_UBOOT_SD_CONFIG=x210_sd_config #  针对 inand 的 uboot 编译配置
BOOTLOADER_UBOOT_NAND_CONFIG=x210_nand_config #针对 nand 的 uboot 编译配置
BOOTLOADER_XBOOT_CONFIG=arm-mpad # xboot 编译配置
KERNEL_NAND_CONFIG=x210_android_nand_defconfig     #针对 nand 的 kernel 配置
KERNEL_INAND_CONFIG=x210_android_inand_defconfig   #针对 inand 的 kernel 配置
FILESYSTEM_CONFIG=PRODUCT-full_x210-eng  #编译 android4.0 时指定开发板型号

export PATH=$SOURCE_DIR/uboot/tools:$PATH      #声明 mkimage 工具的路径
export PATH=$SOURCE_DIR/out/host/linux-x86/bin:$PATH
#计算出 PC 机的 CPU 有几个核，以加快编译速度
CPU_NUM=$(cat /proc/cpuinfo |grep processor|wc -l)
CPU_NUM=$((CPU_NUM+1))
#编译前的一些环境变量的设置
setup_environment()
{
    cd ${SOURCE_DIR};
    mkdir -p ${RELEASE_DIR} || return 1;
    mkdir -p ${TARGET_DIR} || return 1;
    mkdir -p ${TARGET_DIR}/system/app || return 1;
}
#编译针对 uboot 的 bootloader
build_bootloader_uboot_nand()
{
```

```
    cd ${SOURCE_DIR}/uboot || return 1
    make distclean || return 1;
    make ${BOOTLOADER_UBOOT_NAND_CONFIG} || return 1;
    make -j${CPU_NUM}
        mv u-boot.bin uboot_nand.bin

        if [ -f uboot_nand.bin ]; then
        cp uboot_nand.bin ${RELEASE_DIR}/uboot.bin
        echo "^_^ uboot_nand.bin is finished successful!"
        exit
    else
        echo "make error,cann't compile u-boot.bin!"
        exit
    fi

    return 0
}
#编译针对 inand 或 SD 卡的 uboot
build_bootloader_uboot_inand()
{
    cd ${SOURCE_DIR}/uboot || return 1
    make distclean || return 1;
    make ${BOOTLOADER_UBOOT_SD_CONFIG} || return 1;
    make -j${CPU_NUM}
        mv u-boot.bin uboot_inand.bin

        if [ -f uboot_inand.bin ]; then
        cp uboot_inand.bin ${RELEASE_DIR}/uboot.bin
        echo "^_^ uboot_inand.bin is finished successful!"
        exit
    else
        echo "make error,cann't compile u-boot.bin!"
        exit
    fi

    return 0
}
#编译针对 inand 或 sd 卡的 xboot
build_bootloader_xboot()
{
    if [ ! -f ${RELEASE_DIR}/zImage-initrd ]; then
        echo "not found kernel zImage-initrd, please build kernel first">&2
        return 1
```

```
    fi

    if [ ! -f ${RELEASE_DIR}/zImage-android ]; then
        echo "not found kernel zImage-android, please build kernel first">&2
        return 1
    fi

    # copy zImage-initrd and zImage-android to xboot's romdisk directory
    cp -v ${RELEASE_DIR}/zImage-initrd ${SOURCE_DIR}/xboot/src/arch/arm/mach-mpad/romdisk/boot || return 1;
    cp -v ${RELEASE_DIR}/zImage-android ${SOURCE_DIR}/xboot/src/arch/arm/mach-mpad/romdisk/boot || return 1;

    # compiler xboot
    cd ${SOURCE_DIR}/xboot || return 1
    make TARGET=${BOOTLOADER_XBOOT_CONFIG} CROSS=/usr/local/arm/arm-none-eabi-2010-09-51/bin/arm-none-eabi- clean || return 1;
    make TARGET=${BOOTLOADER_XBOOT_CONFIG} CROSS=/usr/local/arm/arm-none-eabi-2010-09-51/bin/arm-none-eabi- || return 1;

    # rm zImage-initrd and zImage-android
    rm -fr ${SOURCE_DIR}/xboot/src/arch/arm/mach-mpad/romdisk/boot/zImage-initrd
    rm -fr ${SOURCE_DIR}/xboot/src/arch/arm/mach-mpad/romdisk/boot/zImage-android

    # copy xboot.bin to release directory
    cp -v ${SOURCE_DIR}/xboot/output/xboot.bin ${RELEASE_DIR}

    echo """>&2
    echo "^_^ xboot path: ${RELEASE_DIR}/xboot.bin">&2
    return 0
}
#编译针对 nand 平台的内核
build_kernel_nand()
{
    cd ${SOURCE_DIR}/kernel || return 1

    make ${KERNEL_NAND_CONFIG} || return 1
    make -j${threads} || return 1
    dd if=${SOURCE_DIR}/kernel/arch/arm/boot/zImage of=${RELEASE_DIR}/zImage bs=2048 count=8192 conv=sync;

    echo """>&2
    echo "^_^ android kernel for nand path: ${RELEASE_DIR}/zImage">&2
```

```
    return 0
}
#编译针对 inand 平台的内核
build_kernel_inand()
{
    cd ${SOURCE_DIR}/kernel || return 1

    make ${KERNEL_INAND_CONFIG} || return 1
    make -j${threads} || return 1
    dd    if=${SOURCE_DIR}/kernel/arch/arm/boot/zImage    of=${RELEASE_DIR}/zImage
bs=2048 count=8192 conv=sync;

    echo """>&2
    echo "^_^ android kernel for inand path: ${RELEASE_DIR}/zImage">&2

    return 0
}
#编译 android4.0.4 文件系统，这时 out/release 目录生成的是 inand 映像
build_system()
{
    # install android busybox
    tar xf ./vendor/9tripod/busybox.tgz -C ${TARGET_DIR}/system/
    # install android app
    cp ./vendor/9tripod/app/*                ${TARGET_DIR}/system/app/ -a

    cd ${SOURCE_DIR} || return 1
    make -j${threads} ${FILESYSTEM_CONFIG} || return 1

    # create android.img.cpio
    rm -fr ${TARGET_DIR}/cpio_list ${TARGET_DIR}/android.img.cpio || { return 1; }
    $TOOLS_DIR/gen_initramfs_list.sh ${TARGET_DIR}/root > ${TARGET_DIR}/cpio_list ||
{ return 1; }
    $TOOLS_DIR/gen_init_cpio              ${TARGET_DIR}/cpio_list              >
${TARGET_DIR}/android.img.cpio || { return 1; }

    # create data.tar
    cd ${TARGET_DIR}/data || { echo "Error: Could not enter the ${TARGET_DIR}/data
directory."; return 1; }
    rm -fr ${TARGET_DIR}/data.tar || { return 1; }
    tar cvf ${TARGET_DIR}/data.tar ./* || { return 1; }

    # create system.tar
```

```
    cd ${TARGET_DIR}/system || { echo "Error: Could not enter the ${TARGET_DIR}/system directory."; return 1; }
    rm -fr ${TARGET_DIR}/system.tar || { return 1; }
    tar cvf ${TARGET_DIR}/system.tar ./* || { return 1; }

    #cp -av ${TARGET_DIR}/installed-files.txt ${RELEASE_DIR}/ || return 1;
    #cp -av ${TARGET_DIR}/installed-files.txt ${RELEASE_DIR}/ || return 1;
    cp -av ${TARGET_DIR}/android.img.cpio ${RELEASE_DIR}/ || return 1;
    cp -av ${TARGET_DIR}/android.img.cpio ${KERNEL_DIR}/ || return 1;
    cp -av ${TARGET_DIR}/system.img ${RELEASE_DIR}/x210.img || return 1;
    #cp -av ${TARGET_DIR}/system.tar ${RELEASE_DIR}/ || return 1;
    cp -av ${TARGET_DIR}/userdata.img ${RELEASE_DIR}/ || return 1;
    #cp -av ${TARGET_DIR}/data.tar ${RELEASE_DIR}/ || return 1;

    echo """>&2
    echo "^_^ system path: ${RELEASE_DIR}/system.tar">&2

    # create uboot_img
    #cd ${TARGET_DIR}
    #echo '****** Make ramdisk image for u-boot ******'
    #mkimage -A arm -O linux -T ramdisk -C none -a 0x30800000 -n "ramdisk" -d ramdisk.img x210-uramdisk.img
    #cp x210-uramdisk.img $RELEASE_DIR/
    #rm -f ramdisk.img

    return 0
}
#打包 android4.0.4 文件系统，这时 out/release 目录生成的是 nand 映像
build_yaffs_image()
{
    rm ./x210_root -rf
    mkdir x210_root

    cp ./out/target/product/x210/system/lib/libstagefright*          ./vendor/9tripod/lib/
    cp ./out/target/product/x210/root/*      ./x210_root/ -a
    cp ./out/target/product/x210/system      ./x210_root/ -a
    #cp ./vendor/9tripod/app/*                ./x210_root/system/app/ -a
    cp ./vendor/9tripod/lib/*                ./x210_root/system/lib/ -a
    cp ./vendor/9tripod/etc/*                ./x210_root/system/etc/ -a
    #cp ./vendor/9tripod/usr/*                ./x210_root/system/usr/ -a
    #cp ./vendor/9tripod/bin/*          ./x210_root/system/bin/ -a

    rm x210_root/init.rc
```

```
    cp device/samsung/x210/init.rc ./x210_root

    chmod 777 ./x210_root/system/vendor/bin/pvrsrvinit
    #chmod 777 ./x210_root/system/bin/bluetoothd
    chmod 777 ./x210_root/system/bin/hciattach
    #chmod 777 ./x210_root/system/bin/pand
    #chmod 777 ./x210_root/system/bin/sdptool
    #rm -rf ./x210_root/system/busybox
    #tar zxvf ./vendor/9tripod/busybox.tgz -C ./x210_root/system/

    mkyaffs2image x210_root/ x210.img

    rm ${RELEASE_DIR}/x210.img
    mv x210.img $RELEASE_DIR/
}
#仅 xboot 使用，将 xboot,文件系统打包成 update.bin 文件
gen_update_bin()
{
    # check image files
    if [ ! -f ${RELEASE_DIR}/xboot.bin ]; then
        echo "not found bootloader xboot.bin, please build bootloader">&2
        return 1
    fi

    if [ ! -f ${RELEASE_DIR}/zImage-initrd ]; then
        echo "not found kernel zImage-initrd, please build kernel first">&2
        return 1
    fi

    if [ ! -f ${RELEASE_DIR}/zImage-android ]; then
        echo "not found kernel zImage-android, please build kernel first">&2
        return 1
    fi

    if [ ! -f ${RELEASE_DIR}/system.tar ]; then
        echo "not found system.tar, please build system">&2
        return 1
    fi

    if [ ! -f ${RELEASE_DIR}/data.tar ]; then
        echo "not found data.tar, please build system">&2
        return 1
    fi
```

```
    rm -fr ${RELEASE_DIR}/tmp || return 1;
    rm -fr ${RELEASE_DIR}/update.bin || return 1;
    mkdir -p ${RELEASE_DIR}/tmp || return 1;

    # copy image files
    cp ${RELEASE_DIR}/xboot.bin ${RELEASE_DIR}/tmp/;
    cp ${RELEASE_DIR}/zImage-initrd ${RELEASE_DIR}/tmp/;
    cp ${RELEASE_DIR}/zImage-android ${RELEASE_DIR}/tmp/;
    cp ${RELEASE_DIR}/system.tar ${RELEASE_DIR}/tmp/;
    cp ${RELEASE_DIR}/data.tar ${RELEASE_DIR}/tmp/;

    # create md5sum.txt
    cd ${RELEASE_DIR}/tmp/;
    find . -type f -print | while read line; do
        if [ $line != 0 ]; then
            md5sum ${line} >> md5sum.txt
        fi
    done

    # genromfs
    genromfs -v -d ${RELEASE_DIR}/tmp/ -f ${RELEASE_DIR}/update.bin || return 1;

    cd ${SOURCE_DIR} || return 1
    rm -fr ${RELEASE_DIR}/tmp || return 1;
    return 0;
}

threads=4;
uboot_nand=no;
uboot_inand=no;
xboot=no;
kernel_nand=no;
kernel_inand=no;
system=no;
update=no;
#接收执行脚本时传入的参数，如果没有传入参数，则默认编译 inand 映像
if [ -z $1 ]; then
    uboot_nand=no
    uboot_inand=yes
    xboot=yes
    kernel_nand=no
        kernel_inand=yes
```

```
        yaffs_system=no
        system=yes
        update=yes
fi
#接收执行脚本时传入的参数，根据不能参数编译不同的映像
while [ "$1" ]; do
        case "$1" in
        -j=*)
                x=$1
                threads=${x#-j=}
                ;;
        -un|--uboot_nand)
                uboot_nand=yes
                uboot_inand=no
                ;;
        -ui|--uboot_inand)
                uboot_inand=yes
                uboot_nand=no
                ;;
        -x|--xboot)
                xboot=yes
                ;;
        -kn|--kernel_nand)
                kernel_nand=yes
                        kernel_inand=no
                ;;
                -ki|--kernel_inand)
                kernel_nand=no
                kernel_inand=yes
                ;;
        -s|--system)
                system=yes
                ;;
        -y|--yaffs2)
                yaffs_system=yes
                ;;
        -U|--update)
                update=yes
                ;;
        -a|--all)
                uboot_nand=no
                uboot_inand=yes
                xboot=yes
```

```
            kernel_nand=no
                    kernel_inand=yes
            yaffs_system=no
            system=yes
            update=yes
            ;;
        -h|--help)
            cat >&2 <<EOF
Usage: build.sh [OPTION]
Build script for compile the source of telechips project.

   -j=n                        using n threads when building source project (example: -j=16)
   -ui, --uboot_inand      build bootloader uboot for sd from source file
   -un, --uboot_nand       build bootloader uboot for nand from source file
   -x, --xboot                 build bootloader xboot from source file
   -kn, --kernel_nand      build kernel for nand flash android and using default config file
   -ki, --kernel_inand     build kernel for inand flash android and using default config file
   -s, --system                build file system from source file
   -y, --yaffs2        build yaffs2 android system
   -U, --update                gen update package update.bin
   -a, --all               build all, include anything
   -h, --help                  display this help and exit
EOF
            exit 0
            ;;
        *)
            echo "mk: Unrecognised option $1">&2
            exit 1
            ;;
        esac
        shift
done

setup_environment || exit 1
#根据前面得到的参数编译指定的映像
if [ "${system}" = yes ]; then
        build_system || exit 1
fi

if [ "${yaffs_system}" = yes ]; then
        build_yaffs_image || exit 1
fi
```

```
if [ "${kernel_nand}" = yes ]; then
        build_kernel_nand || exit 1
fi

if [ "${kernel_inand}" = yes ]; then
        build_kernel_inand || exit 1
fi

if [ "${uboot_nand}" = yes ]; then
        build_bootloader_uboot_nand || exit 1
fi

if [ "${uboot_inand}" = yes ]; then
        build_bootloader_uboot_inand || exit 1
fi

if [ "${xboot}" = yes ]; then
        build_bootloader_xboot || exit 1
fi

if [ "${update}" = yes ]; then
        gen_update_bin || exit 1
fi

exit 0
```

4.2 android.img.cpio 解压和打包脚本

当映像存放在ＳＤ卡中时，android 的初始化脚本 init.rc 是会打包在 android.img.cpio 文件中的。如果我们需要修改 init.rc，需要首先修改 android 源码，再重新编译，这样会浪费大量的时间。其实，我们可以直接解压 android.img.cpio 文件，手动修改 init.rc 文件后，再压缩打包，最后将它整合到内核即可。为此，我们编写了解压和压缩 android.img.cpio 的脚本。具体使用方法如下：

在 android 源码根目录下，执行./android_cpio.sh，弹出如下界面：

```
lqm@lqm:~/android_gingerbread_v10/android$ ./android_cpio.sh
Modify the android.img.cpio
1.unzip the image
2.Create the image
3.exit
Choose:
```

输入 1，回车，即将 out/release 目录下的 android.img.cpio 解压出来了，解压路径为：

```
out/release/tmp
```

这时，就可以手动的修改 init.rc 文件了。修改完成后，再次执行该脚本，输入 2，回车，即可将 out/release/tmp 目录下的文件打包成 android.img.cpio，同时删掉 out/release/tmp 目录。

脚本内容即详细解释如下：

```
#!/bin/bash
# create: liuqiming
# date:     2011-12-16
# mail:     phosphor88@163.com

#在终端打印提示信息
echo "Modify the android.img.cpio"
echo "1.unzip the image"
echo "2.Create the image"
echo "3.exit"
#声明环境变量
SOURCE_DIR=$(cd `dirname $0` ; pwd)
TOOLS_DIR=${SOURCE_DIR}/tools
TARGET_DIR=${SOURCE_DIR}/out/release
#读取输入的键值
read -p "Choose:" CHOOSE
#根据输入的键值进行相关操作
if [ "1" = ${CHOOSE} ];then
        echo "unzip android.img.cpio"
        cd ${TARGET_DIR}
        [ -e "tmp" ] ||{ echo "mkdir tmp"; mkdir tmp;}
        [ -e "android.img.cpio" ] || { echo "error!can't find andaroid.img.cpio!"; exit; }
        cd tmp
        #解压 cpio 文件包
        cpio -idmv --no-absolute-filenames < ../android.img.cpio
        echo "^_^ unzip android.img.cpio finished!"
        exit

elif [ "2" = ${CHOOSE} ];then
        echo "create android.img.cpio test"
        [ -e "${TARGET_DIR}/tmp" ] || { echo "can't find [tmp],please unzip android.img.cpio
first!"; exit; }
        rm -f ${TARGET_DIR}/cpio_list
        rm -f ${TARGET_DIR}/android.img.cpio
        #打包 cpio 文件包
        $TOOLS_DIR/gen_initramfs_list.sh ${TARGET_DIR}/tmp > ${TARGET_DIR}/cpio_list ||
{ exit; }
        $TOOLS_DIR/gen_init_cpio                ${TARGET_DIR}/cpio_list                >
${TARGET_DIR}/android.img.cpio || { exit; }
        rm -rf ${TARGET_DIR}/tmp
        echo "^_^ Create android.img.cpio finished!"
        exit
```

```
elif [ "3" = ${CHOOSE} ];then
        exit
fi
```

第5章 源码编译

x210v3s 开发板可选配 inand 或 nand flash 的存储芯片，二者程序是有区别的，但是共用一个源码包管理。因此，在编译时，需要指定编译类型。

说明：编译映像时一定要使用普通权限编译。不论编译针对 inand 还是 nand flash 的映像文件，最终都只需要三个映像文件：

- uboot.bin: bootloader，用于引导内核
- zImage-android：内核
- x210.img：android 文件系统
- xboot.bin:bootloader，用于引导内核
- android-update.bin：基于 xboot 的单一更新文件

5.1 编译针对 inand 的 uboot

在 android 源码目录下执行如下命令编译 uboot，编译完成后映像文件 uboot.bin 会释放到 out/release 目录。

```
./mk      -ui
```

5.2 编译针对 nand 的 uboot

在 android 源码目录下执行如下命令编译 uboot，编译完成后映像文件 uboot.bin 会释放到 out/release 目录。

```
./mk      -un
```

5.3 编译 xboot

xboot 仅支持 inand 平台。在 android 源码目录下执行如下命令编译 xboot，编译完成后映像文件 xboot.bin 会释放到 out/release 目录。

```
./mk   -x
```

5.4 编译 i210 平台的 xboot

将光盘 B 中 android4.0.4\image\inand\i210\ gpio-button.c 文件覆盖到如下目录：

xboot/src/arch/arm32/mach-x210v3s/driver

再执行如下指令编译 xboot:

```
./mk   -x
```

5.5 编译针对 inand 的 android 内核

在 android 源码目录下执行如下命令编译 android 内核，编译完成后映像文件 zImage 会释放到 out/release 目录。

```
./mk      -ki
```

5.6 编译 i210 平台针对 inand 的 android 内核

进入 android4.0 内核的如下目录：

kernel/arch/arm/configs

删除 x210_android_inand_defconfig 文件，将 i210_android_inand_defconfig 重命名为 x210_android_inand_defconfig，再执行如下指令编译内核：

```
./mk       -ki
```

5.7 编译针对 nand 的 android 内核

在 android 源码目录下执行如下命令编译 android 内核，编译完成后映像文件 zImage 会释放到 out/release 目录。

```
./mk       -kn
```

5.8 编译 i210 平台针对 nand 的内核

进入 android4.0 内核的如下目录：

kernel/arch/arm/configs

删除 x210_android_nand_defconfig 文件，将 i210_android_nand_defconfig 重命名为 x210_android_nand_defconfig，再执行如下指令编译内核：

```
./mk       -kn
```

5.9 编译针对 inand 的 android 文件系统

首先，确定 android 根目录下是否存在 out 目录，第一次编译时，删掉整个 out 目录，在 android 源码目录下执行如下命令编译 android 映像文件，编译完成后映像文件 x210.img 会释放到 out/release 目录。

```
./mk       -s
```

5.10 编译针对 nand 的 android 文件系统

编译完针对 inand 的 android 文件系统后，其实针对 nand 的文件系统也已经编译好了，我们只需要打包即可，但是注意需要将必要的文件做少许修改。这里我们已经全集成到脚本中，执行如下指令：

```
./mk       -y
```

打包完成后映像文件 x210.img 会释放到 out/release 目录，替换掉 inand 的映像。

第6章 映像文件的烧写

6.1 ubuntu 下 fastboot 的安装

6.1.1 安装 fastboot

在编译完 android 文件系统后，将会在 out/host/linux-x86/bin 目录下生成 fastboot 文件。将 fastboot 文件拷贝到 ubuntu 的/sbin 目录，或者将 out/host/linux-x86/bin 这个目录声明到~/.bashrc 文件中，就完成 fastboot 安装了。

6.1.2 新建 51-android.rules

新建 51-android.rules 文件，内容如下：

```
# adb protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e12", MODE="0666",
OWNER="terry"
# fastboot protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff", MODE="0666",
OWNER="terry"
# adb protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e22", MODE="0666",
OWNER="terry"
# fastboot protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e20", MODE="0666",
OWNER="terry"
# adb protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", ATTR{idProduct}=="70a9", MODE="0666",
OWNER="terry"
# fastboot protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="708c", MODE="0666",
OWNER="terry"
# adb protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", ATTR{idProduct}=="6860", MODE="0666",
OWNER="terry"
# fastboot protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e30", MODE="0666",
OWNER="terry"
# adb protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d101", MODE="0666",
OWNER="terry"
# fastboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d022", MODE="0666",
OWNER="terry"
# usbboot protocol on panda (PandaBoard)
```

```
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d010", MODE="0666",
OWNER="terry"
# fastboot protocol on x210
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="0002", MODE="0666",
OWNER="terry"
# usb->uart
SUBSYSTEM=="usb", ATTR{idVendor}=="067b", ATTR{idProduct}=="2303", MODE="0600",
OWNER="terry"
```

注意，OWNER 里面填的"terry"务必换成自己 ubuntu 系统的用户名。之后将 51-android.rules 文件复制到/etc/udev/rules.d/ 目录下。

到此，就可以使用 fastboot 更新映像了。如果不建 51-android.rules 这个文件，使用 fastboot 更新时需要使用 root 权限。

6.2 ubuntu 下烧写映像文件到 nand flash

在 nand flash 中没有任何程序时，在 ubuntu 或其他 linux 系统下烧写映像文件到 nand flash，具体步骤如下：

第一步：进入 android 源码根目录，执行如下脚本编译 uboot

```
./mk -un
```

编译完成后，会在 uboot 目录下生成 uboot_nand.bin 文件。

注意，如果需要烧写映像到 nand flash 中，这里一定要生成 uboot_nand.bin 文件，如果生成了 uboot_inand.bin 文件，映像将写能烧写到 inand 卡中，详细请分析 uboot 目录下的 mk 脚本。

第二步：将 SD 卡放到读卡器上，再将读卡器插到运行 linux 系统的 PC 机上，初次使用 SD 卡时，需将 SD 卡的分区全部删掉，执行如下指令：

```
sudo fdisk /dev/sdb
```

注意，多数情况下 SD 卡的盘符为 sdb，少数情况下会出现 sdc,sde 等盘符，具体需使用 cat /proc/partition 指令查询。

再输入 d，回车，如果存在多个盘符，按照提示逐个删掉，删完后按 w 保存，再重插读卡器，使用 ls /dev/sd*指令查询，这时 sdb 应该只存在一个，没有多余的分区了。进入 sd_fusing 目录并执行脚本烧录 SD 卡映像：

```
cd uboot/sd_fusing/
```

首次烧录时，查询在 uboot/sd_fusing 目录下是否存在 mkbl1 文件，如果不存在，在当前目录下 make 一下，就生成了。再执行如下指令

```
sudo bash ./nand_fusing.sh /dev/sdb
```

这时，uboot 就烧到 SD 卡中了。

第三步：将 SD 卡插到 x210v3s 开发板的右侧卡槽，将拨码开关拨至从 SD 卡启动，再启动开发板，这时可以从串口终端看到 uboot 已经启动了。在 3 秒倒计时之内按下空格键，进入控制台终端，通过 miniUSB 延长线将开发板连接到 PC 机，输入 fastboot，打印信息如下：

```
U-Boot 1.3.4-dirty (Apr  8 2012 - 11:40:21) for SMDKV210


CPU:   S5PV210@1000MHz(OK)
```

```
        APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
        MPLL = 667MHz, EPLL = 96MHz
                        HclkDsys = 166MHz, PclkDsys = 83MHz
                        HclkPsys = 133MHz, PclkPsys = 66MHz
                        SCLKA2M  = 200MHz
Serial = CLKUART
Board:   X210
DRAM:    512 MB
Flash:   8 MB
SD/MMC: Card init fail!
0 MB
NAND:    512 MB
*** Warning - using default environment

In:      serial
Out:     serial
Err:     serial
checking mode for fastboot ...
Hit any key to stop autoboot:  0
SMDKV210 # fastboot
Fastboot: employ default partition information
[Partition table on NAND]
ptn 0 name='bootloader' start=0x0 len=0x100000(~1024KB)
ptn 1 name='recovery' start=0x100000 len=0x500000(~5120KB)
ptn 2 name='kernel' start=0x600000 len=0x500000(~5120KB)
ptn 3 name='ramdisk' start=0xB00000 len=0x300000(~3072KB)
ptn 4 name='system' start=0xE00000 len=0x8200000(~133120KB) (Yaffs)
ptn 5 name='cache' start=0x9000000 len=0x3C00000(~61440KB) (Yaffs)
ptn 6 name='userdata' start=0xCC00000 len=N/A (Yaffs)
```

第四步：另外开启一个命令行终端，输入如下指令：

```
lqm@lqm:~$ fastboot devices
SMDKC110-01      fastboot
lqm@lqm:~$
```

如果弹出上面的信息，表明已经找到设备，再通过如下指令烧写 uboot:

```
cd android_gingerbread_v10/android/out/release/
fastboot flash bootloader uboot.bin
```

打印信息如下：

```
lqm@lqm:~/android_gingerbread_v10/android/out/release$ fastboot flash bootloader uboot.bin
sending 'bootloader' (320 KB)...
OKAY [  0.086s]
writing 'bootloader'...
OKAY [  0.333s]
```

```
finished. total time: 0.419s
lqm@lqm:~/android_gingerbread_v10/android/out/release$
```

表示烧录成功。

第五步：将开发板拨码开关切回 nand 启动，开机，发现去掉 S D 卡后，也能启动了。说明 uboot 已经成功更新到 nand flash 中。同样，在控制终端输入 fastboot，然后在另外一个终端进入 out/release 目录，执行如下指令烧写剩下的映像文件：

```
fastboot flash kernel zImage
fastboot flash system x210.img
```

6.3 windows 下烧写映像文件到 nand flash

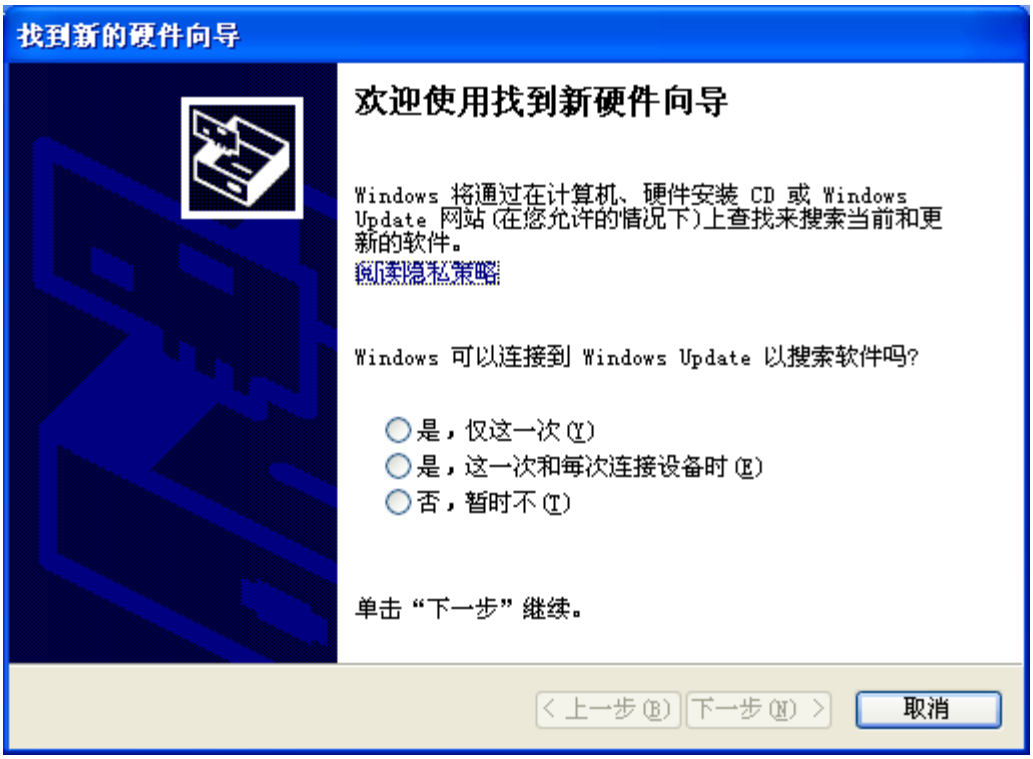
说明:下文提供了三种烧写映像文件到 nand flash 的方法，强烈推荐大家使用 fastboot 烧写各种映像。由于每个程序员环境不太一样，因此特别公布使用 DNW 和网口的烧录方法，仅提供参考。在 windows 平台下，第一次烧写 nand flash 时，可以通过 DNW+USB 将 uboot 固化到 nand flash 中，随后就可以使用强大的 fastboot 更新其余所有映像了。在 ubuntu 平台下，第一次烧写 nand flash 时，可以先将 uboot 烧写到 SD 卡上，再将开发板切换成 SD 卡启动，再通过 SD 卡上运行起来的 uboot，使用其 fastboot 功能更新剩余的映像。

6.3.1 在 windows 下使用 DNW 烧写 uboot

说明：使用 DNW 烧写映像，我们只在 Windows XP 下测试通过，并没有在 Windows7,Windows8 等上测试，光盘中提供的驱动也仅支持 Windows XP。

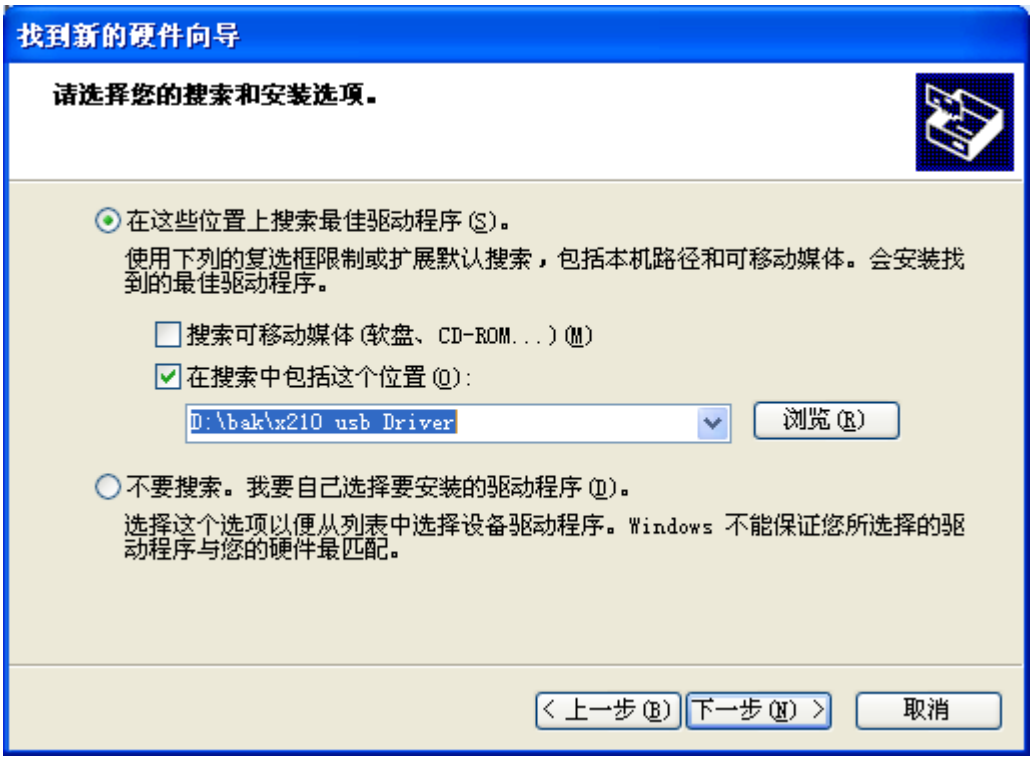
第一步：烧写 x210_usb.bin 文件，初始化 210 处理器相关寄存器

将跳线的 OM5 设置为 ON 状态，使用标准串口线连接 PC 机和开发板，使用 USB 延长线（标准 USB 头转 miniUSB 头）连接开发板的 USB Device 接口，打开电源总开关，同时按住 POWER 键不放，这时，PC 机会自动检测到有新硬件，并要求安装驱动。



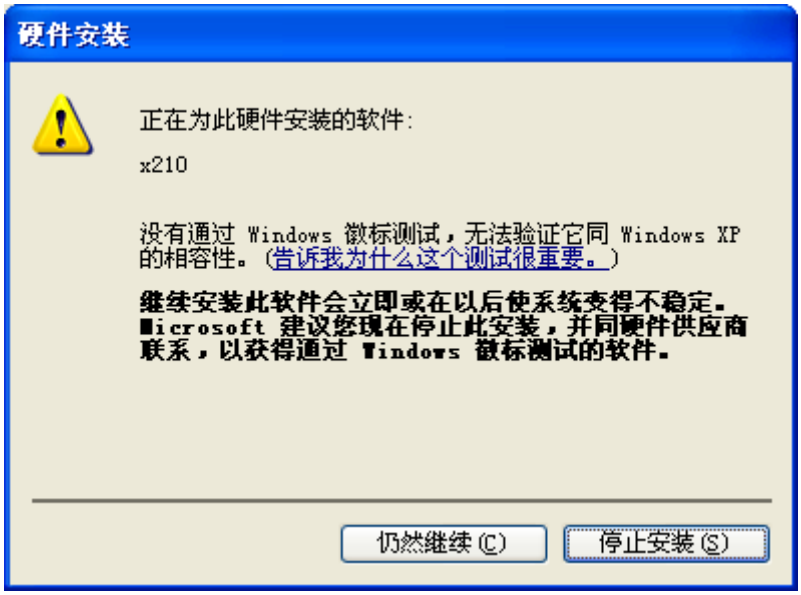
点击否，暂时不，下一步

选择从列表或指定位置安装，下一步



选择第一项，选中在搜索中包括这个位置，点击浏览，找到 x210 USB 驱动目录，点击下一步：

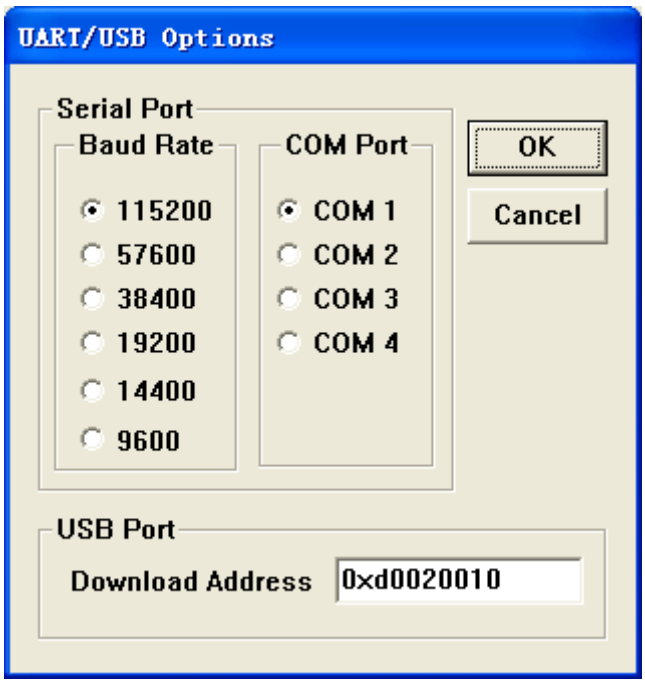
如果之前安装过三星平台的 DNW 驱动，这里可能会有多项选项，找到 x210 所在选项，点击下一步：



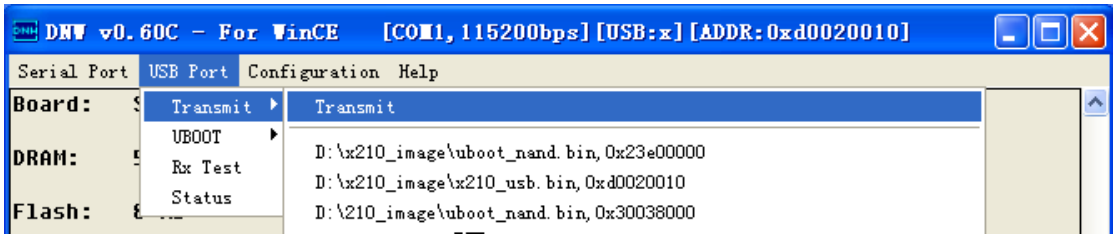
提示如上警告时，点击仍然继续：

安装完毕，DNW 上的 USB 栏就会有 OK 的显示。

第一次下载 x210_usb.bin 时，下载地址设置为 0xd0020010，



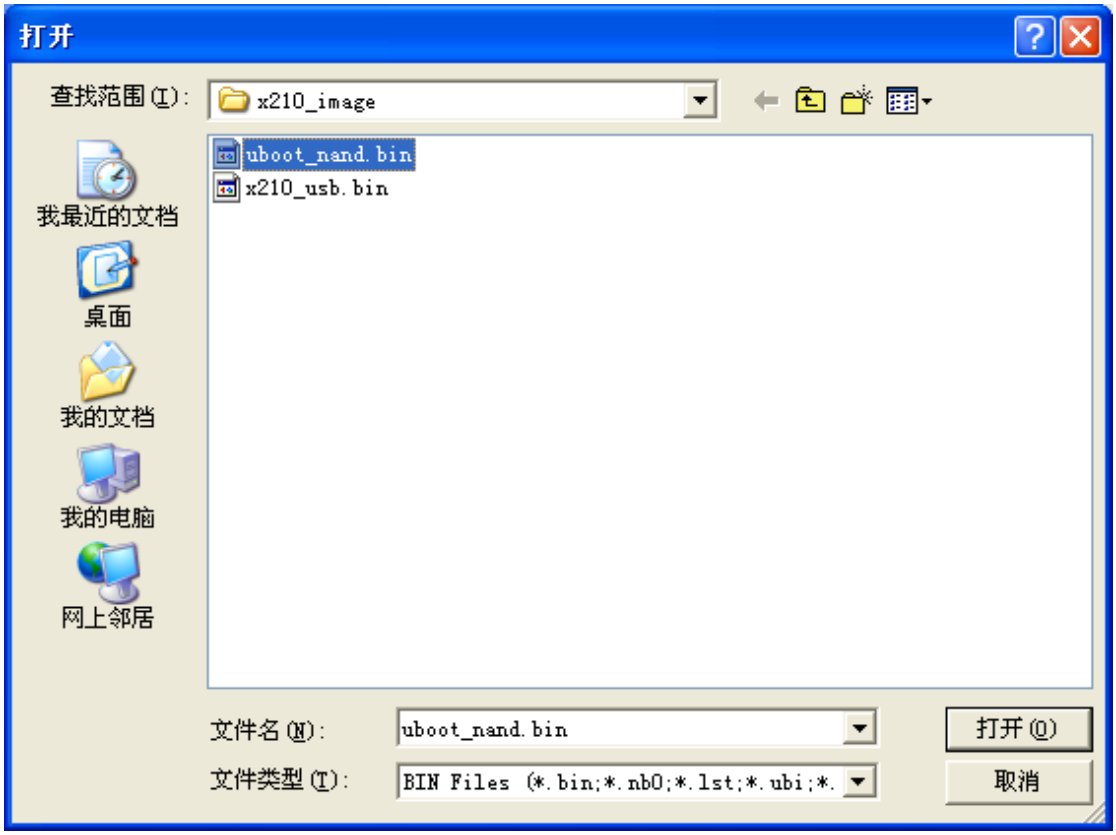
点击 USB Port->Transmit->Transmit

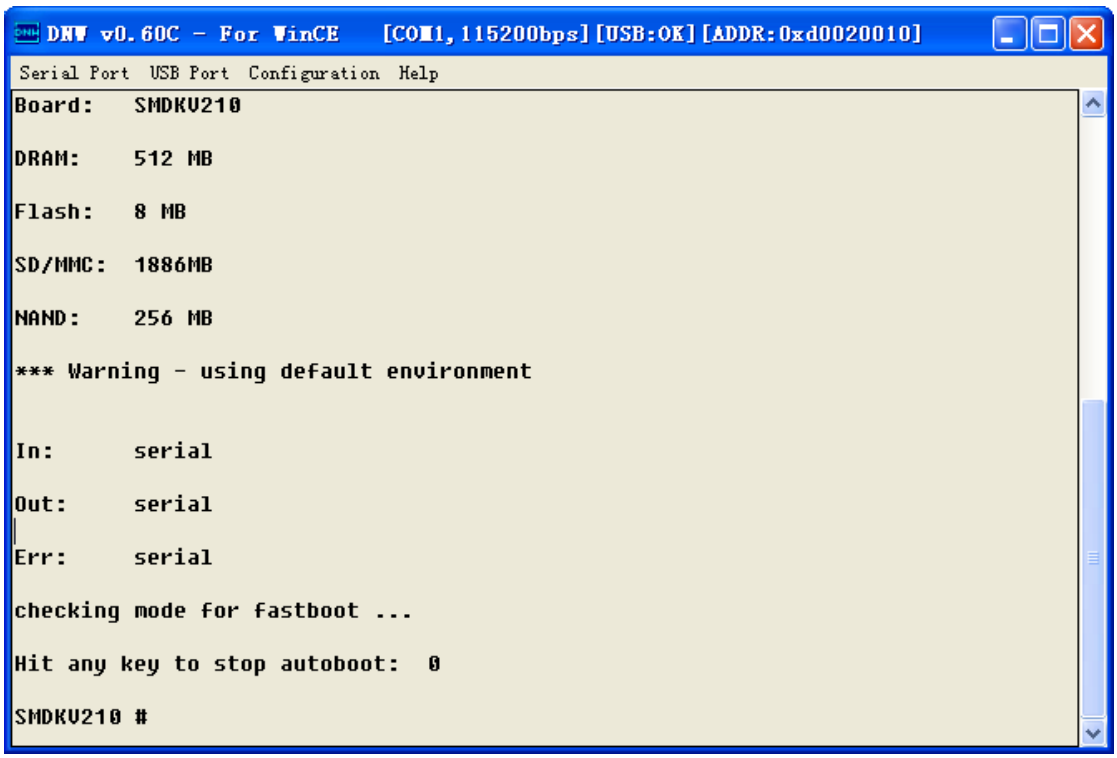


选择 x210_usb.bin，这时 DNW 顶部的 USB 状态会由 OK 变为 x，再又会迅速变回 OK，表示 x210_usb.bin 下载成功，这时，210 相关寄存器已经初始化完毕，接下来就可以下载 uboot 到内存中运行了。

第二步：下载 uboot.bin 文件到 RAM 运行

将下载地址设置为 0x23e00000，再点击 USB Port->Transmit->Transmit，找到 uboot.bin 映像文件：



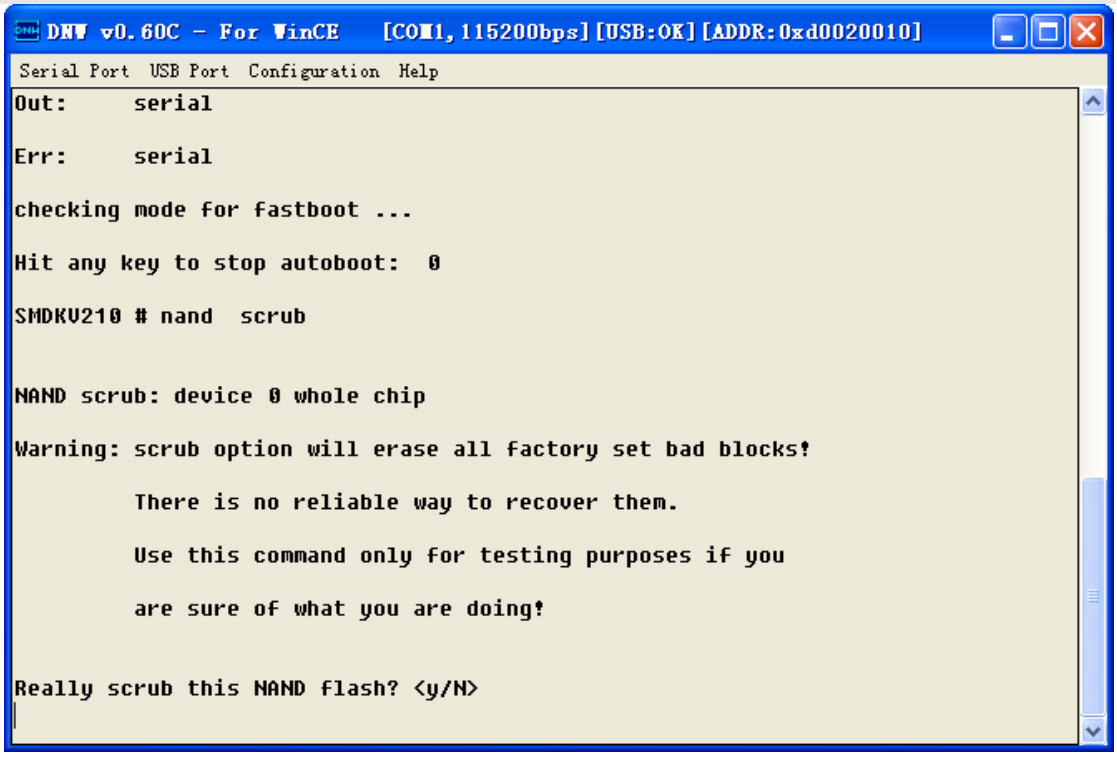


可以看到，uboot 已经运行起来了。注意，这时程序已经软置锁，即到这个时候，我们可以松开开发板的 POWER 键了。同时，程序只是在 ram 中运行，并没有写到 nand flash 中.

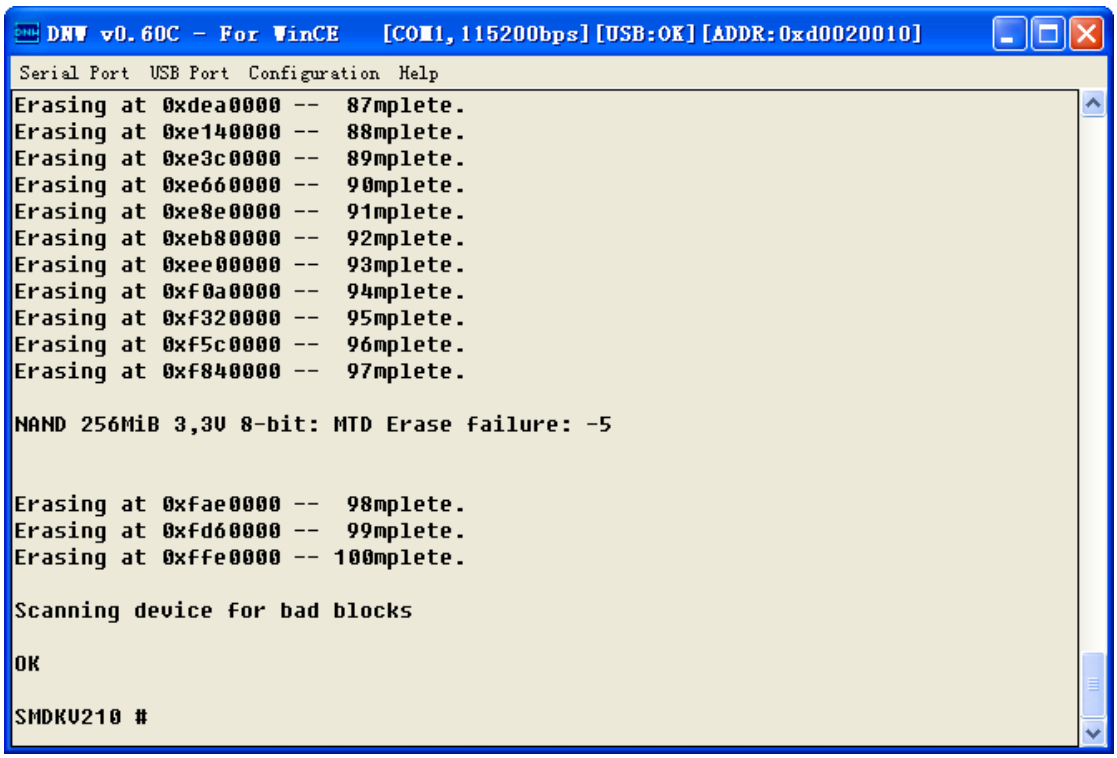
第三步：格式化 nand flash

执行如下命令，格式化整个 nand flash:

```
nand  scrub
```



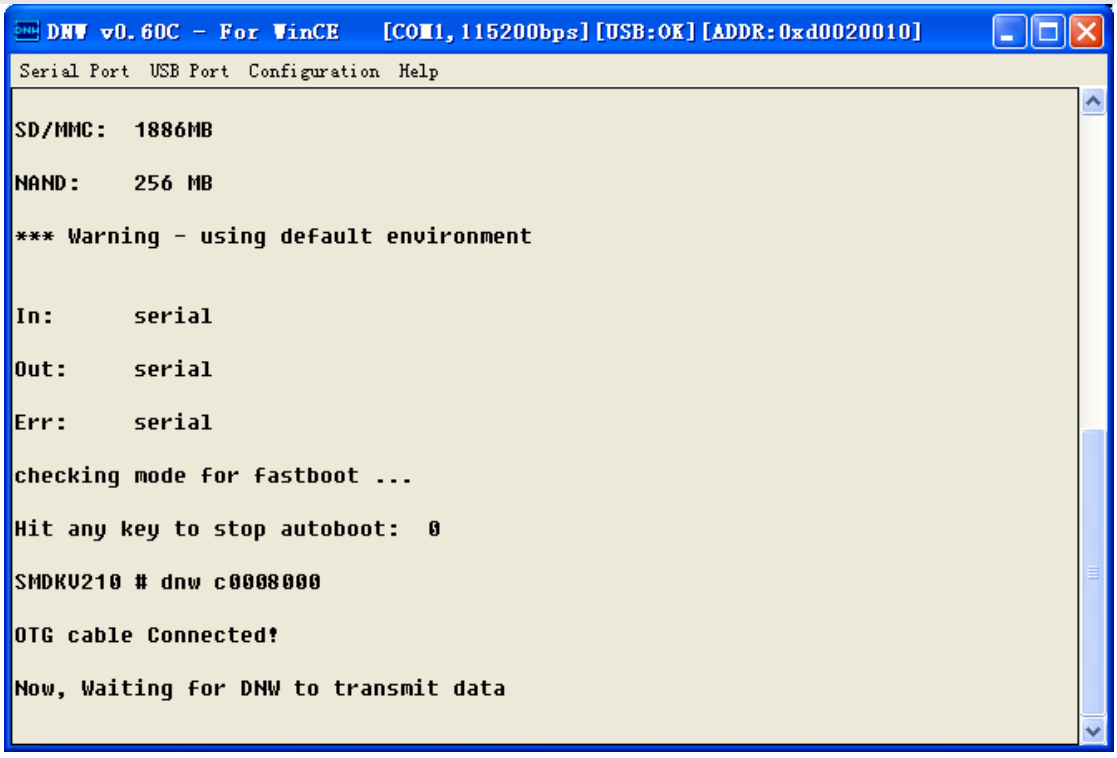
选择 y，回车



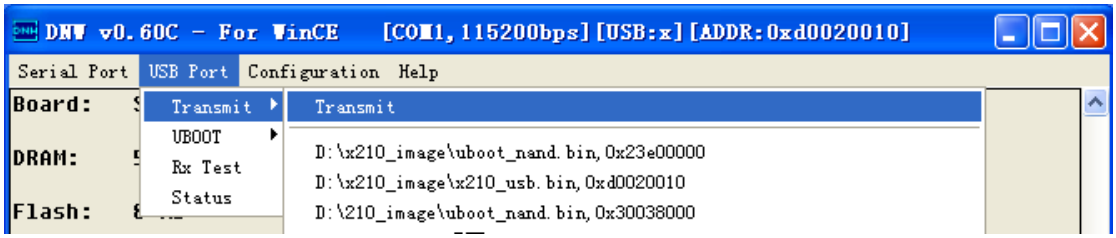
第四步：下载 uboot.bin 到 nand flash

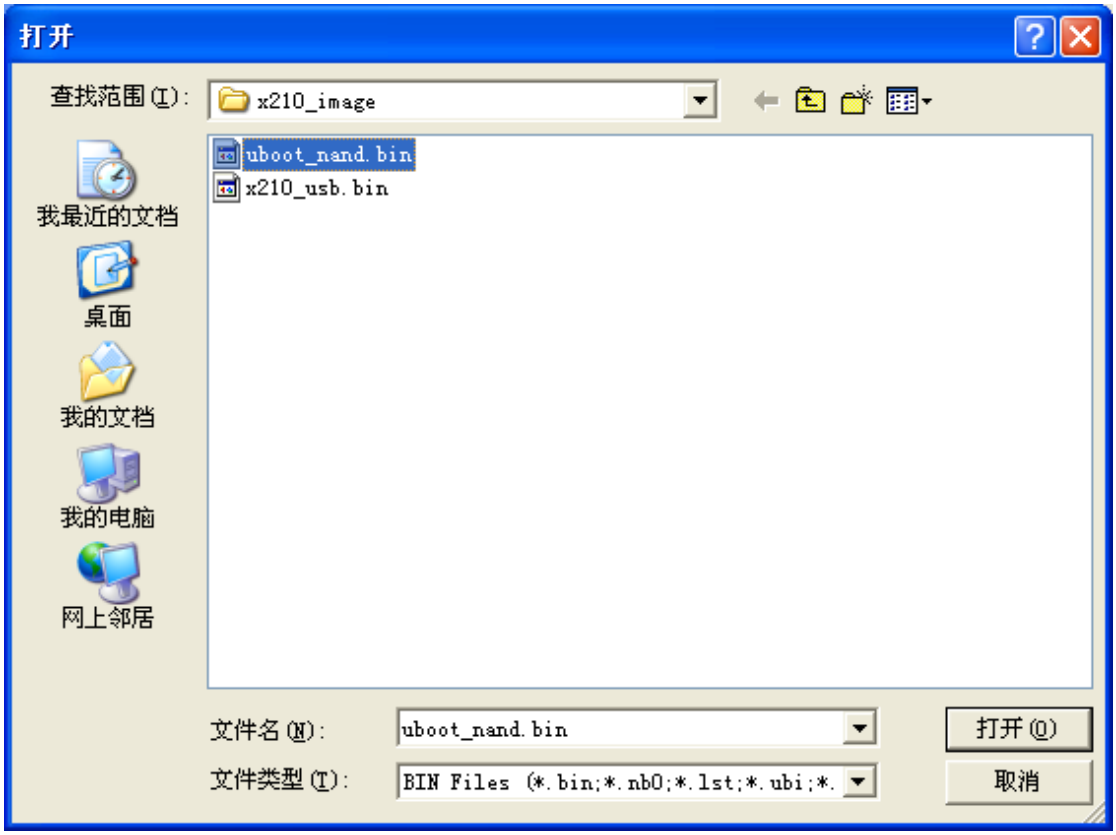
输入如下指令：

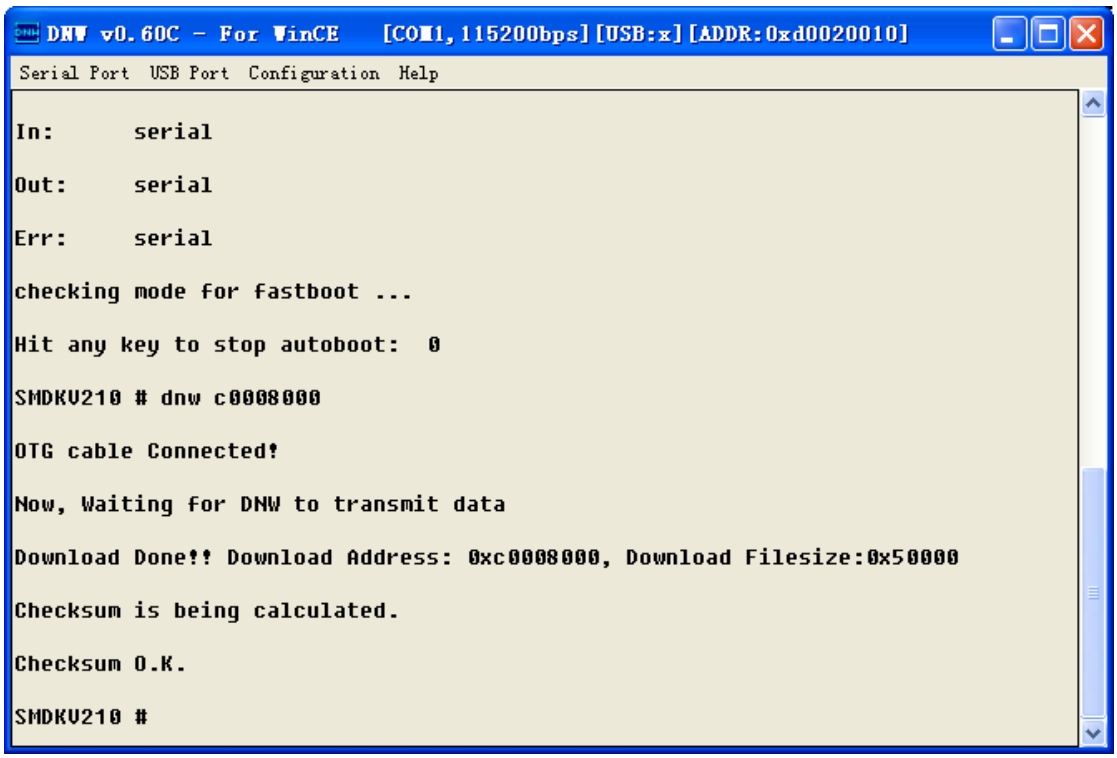
```
dnw    c0008000
```



下载 uboot.bin

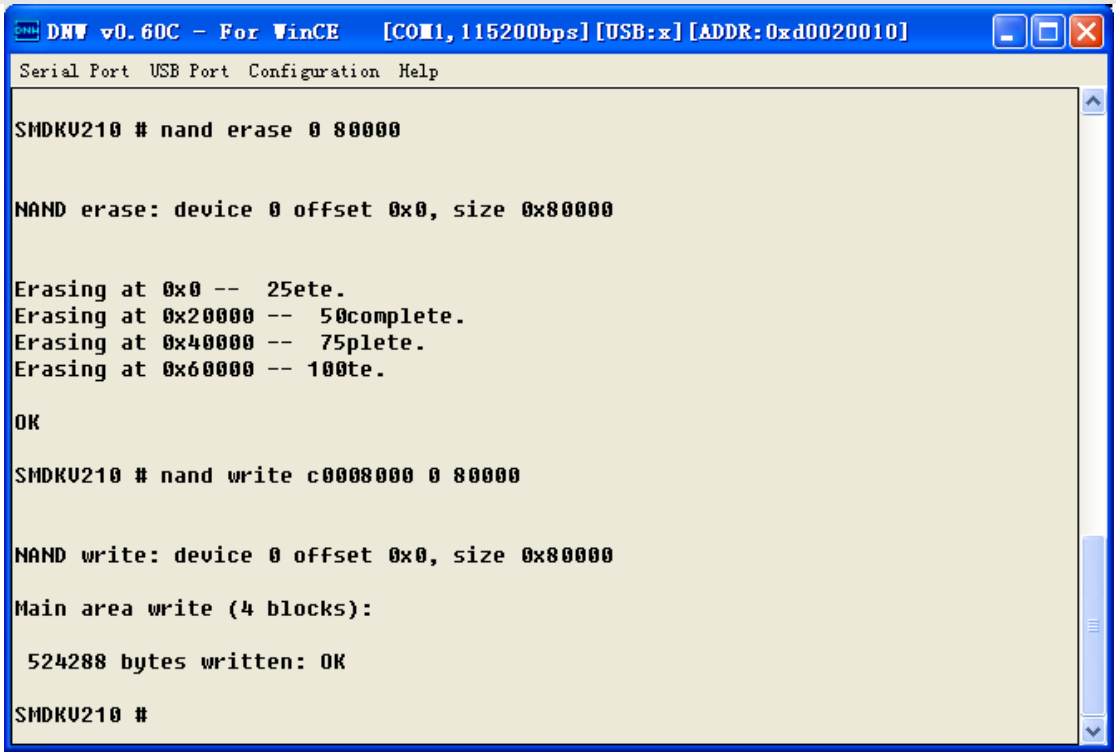

DNW v0.60C - For WinCE [COM1,115200bps][USB:x][ADDR:0xd0020010]
Serial Port
USB Port
Configuration
Help
Board:
DRAM:
Flash:
Transmit
UBOOT
Rx Test
Status
Transmit
D:\x210_image\uboot_nand.bin,0x23e00000
D:\x210_image\x210_usb.bin,0xd0020010
D:\210_image\uboot_nand.bin,0x30038000


打开
查找范围(I):
x210_image
uboot_nand.bin
x210_usb.bin
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
uboot_nand.bin
打开(O)
文件类型(T):
BIN Files (*.bin;*.nb0;*.lst;*.ubi;*.
取消



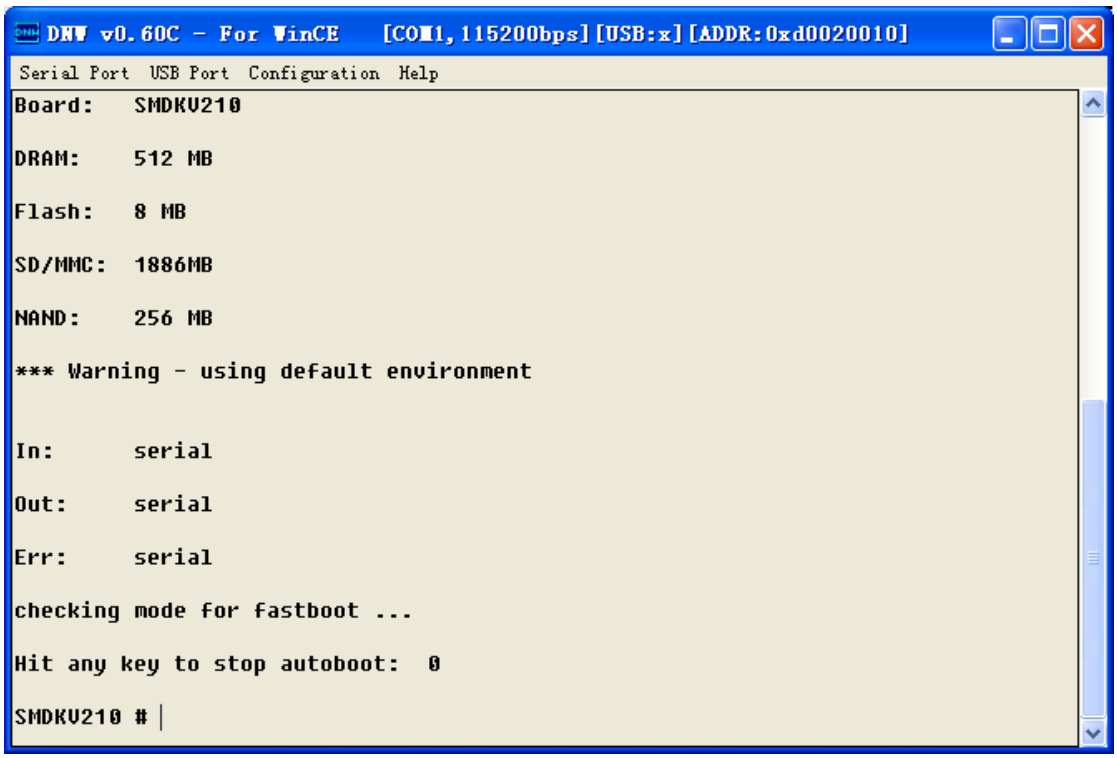
将 uboot.bin 写入 nand flash:

```
nand erase    0    80000
nand write    c0008000    0    80000
```



这时，uboot 已经成功写入 nand flash。

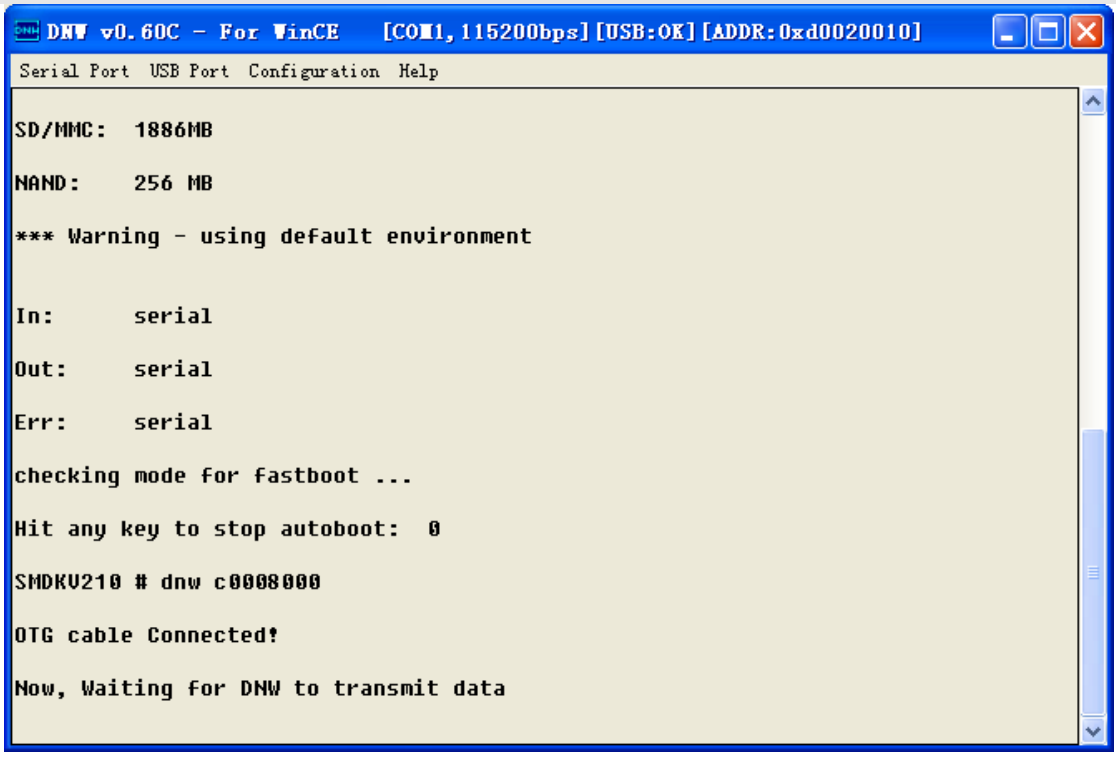
第五步：将拨码开关拨至 nand flash 启动，重新开机，就可以看到 uboot 已经运行起来了。



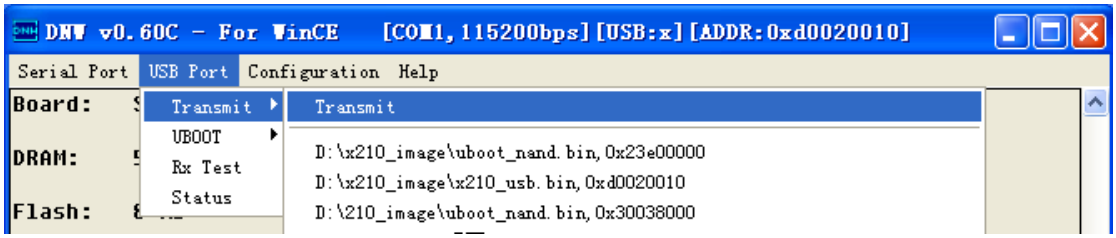
6.3.2 在 windows 下使用 DNW 烧写 kernel

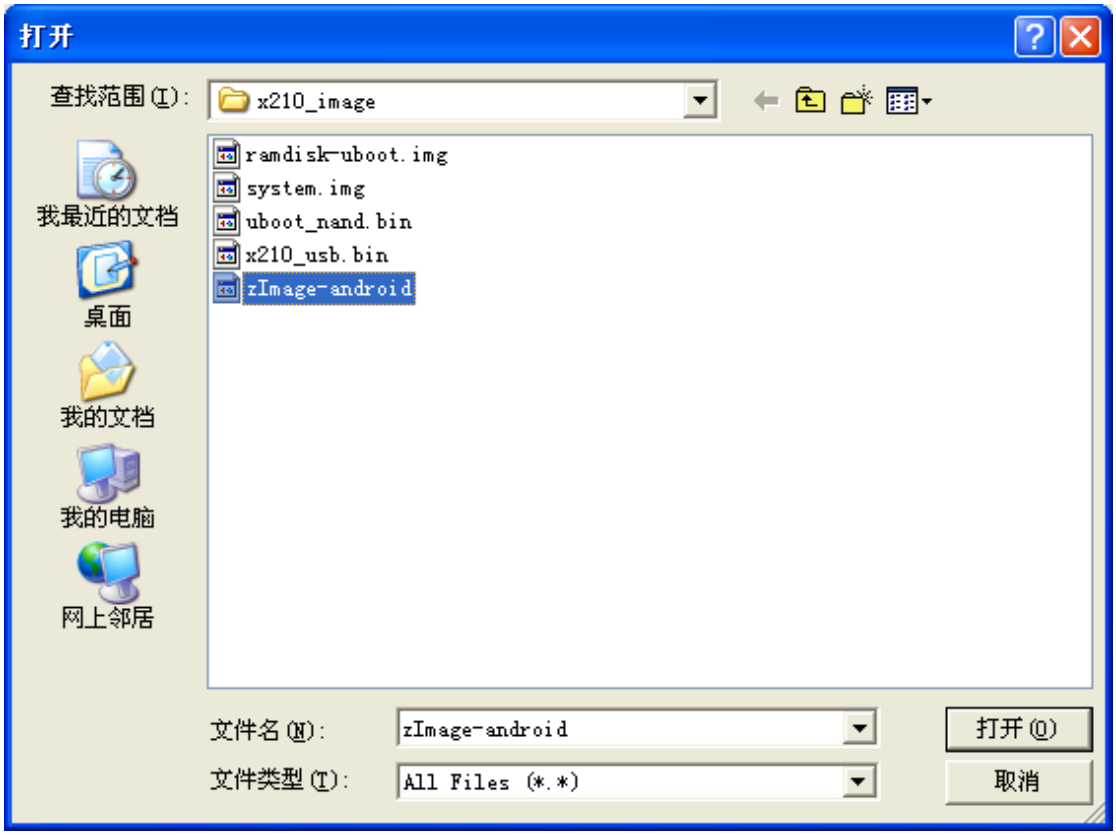
uboot 运行起来后，输入如下指令：

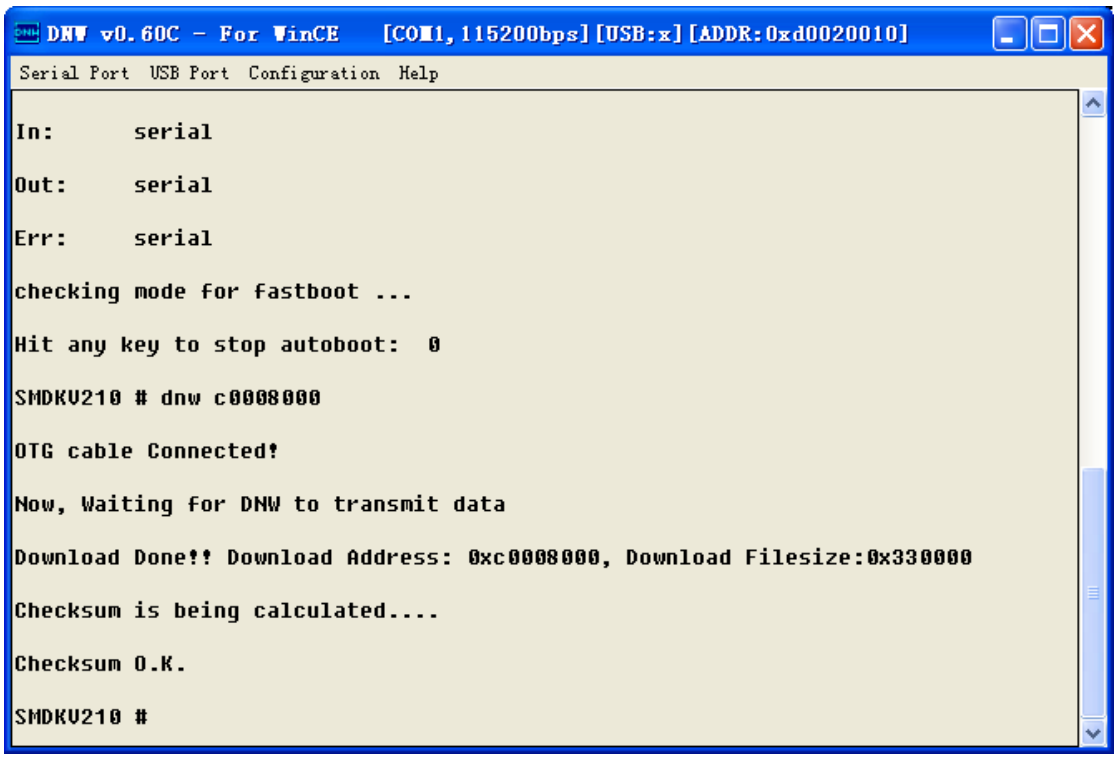
```
dnw  c0008000
```



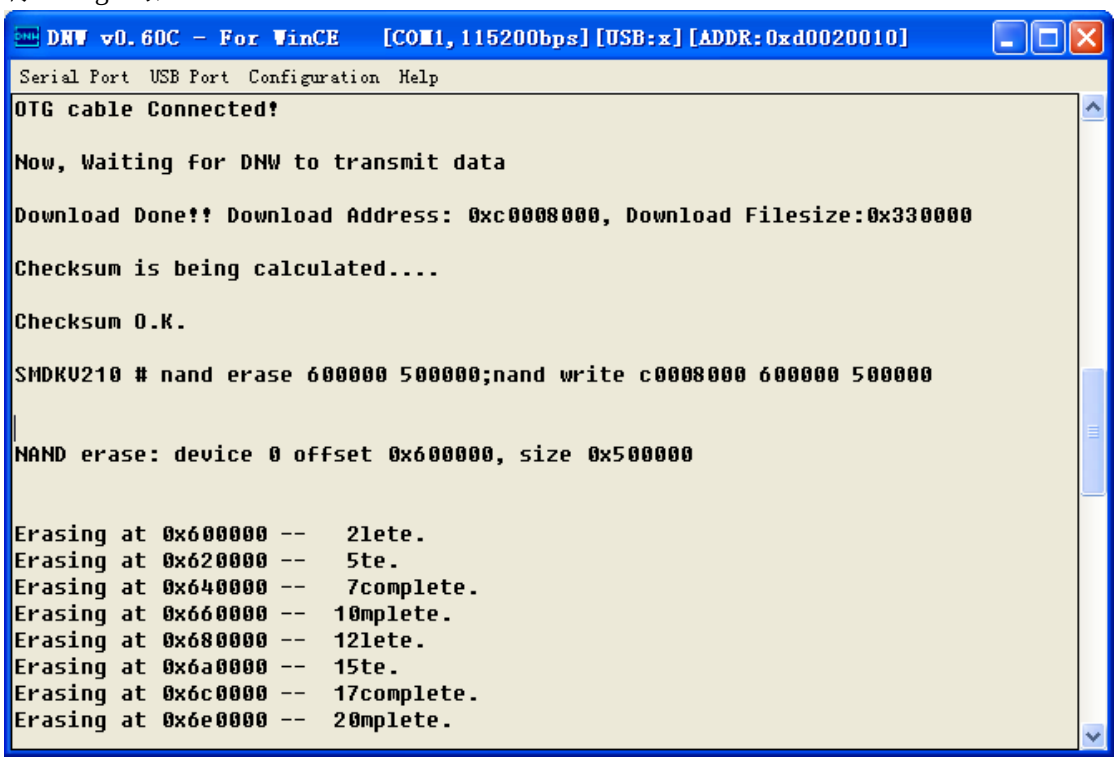
将 zImage-android 下载到 ram：

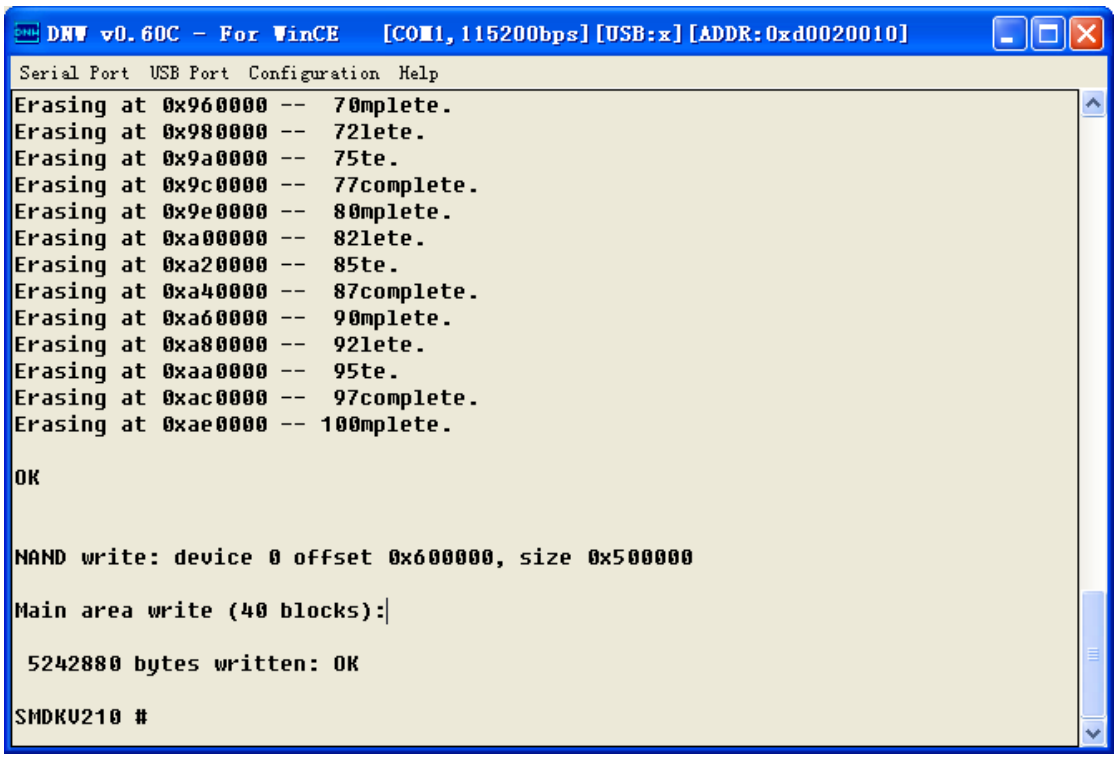

DNW v0.60C - For WinCE [COM1,115200bps][USB:x][ADDR:0xd0020010]
Serial Port
USB Port
Configuration
Help
Board:
DRAM:
Flash:
Transmit
UBOOT
Rx Test
Status
Transmit
D:\x210_image\uboot_nand.bin,0x23e00000
D:\x210_image\x210_usb.bin,0xd0020010
D:\210_image\uboot_nand.bin,0x30038000


打开
查找范围(I):
x210_image
ramdisk-uboot.img
system.img
uboot_nand.bin
x210_usb.bin
zImage-android
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
zImage-android
文件类型(T):
All Files (*.*)
打开(O)
取消

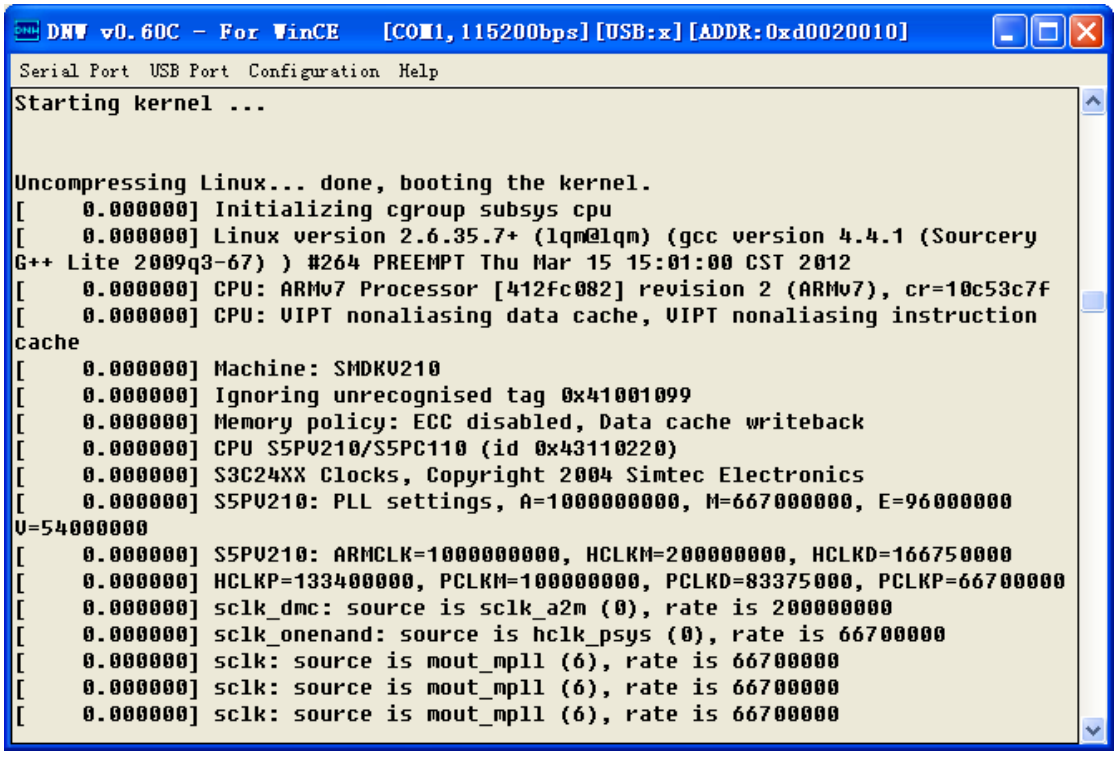


将 zImage 写入 nand flash：





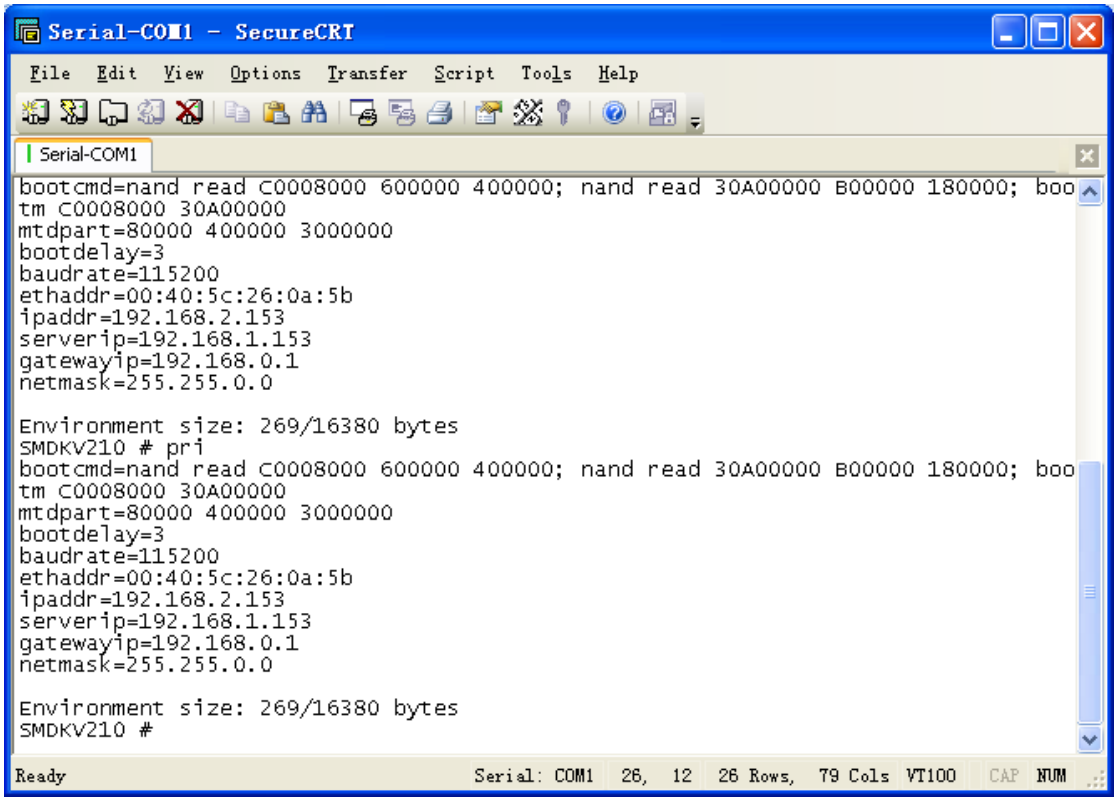
重启开发板，可以看到内核已经运行起来了。



6.3.3 在 windows 下使用网口更新 uboot

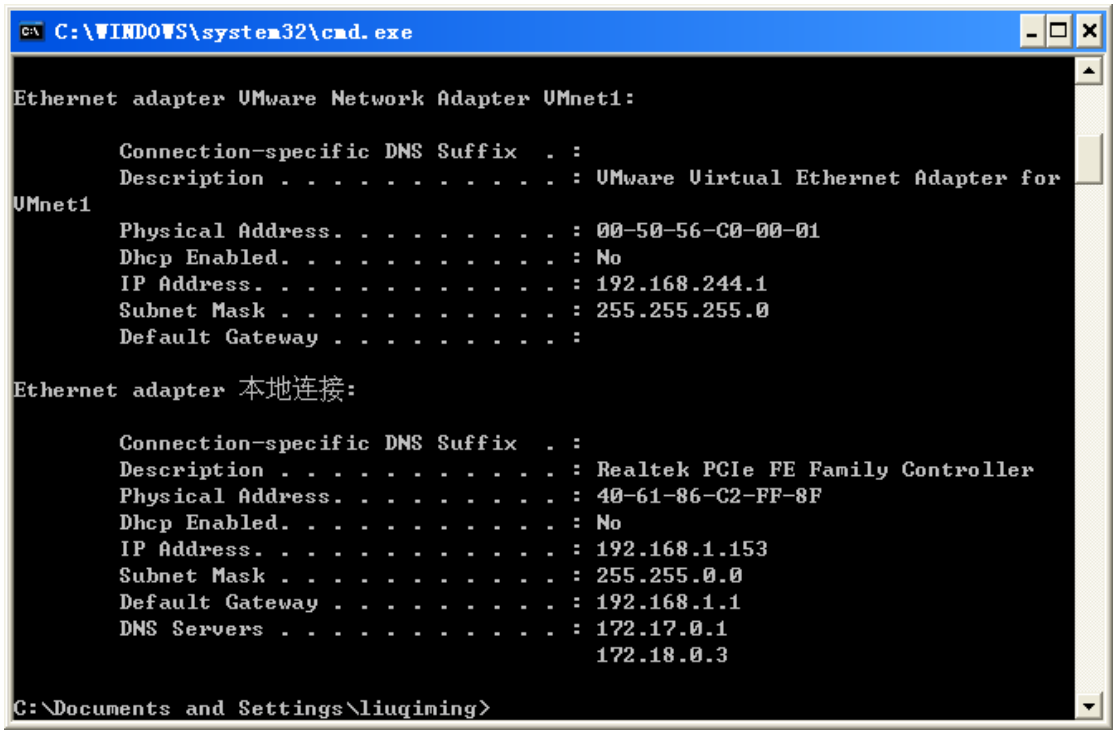
说明：由于使用网口下载映像时，无须用到 USB，因此这里我们使用 SecureCRT 串口工具操作，不再使用 DNW。

进入 uboot 命令行，使用 printenv 命令查看当前网络配置：



在 windows xp 下，输入 windows+r，输入 cmd，进入 DOS 命令行，输入如下命令查询 PC 机的 IP 地址：

```
ipconfig /all
```

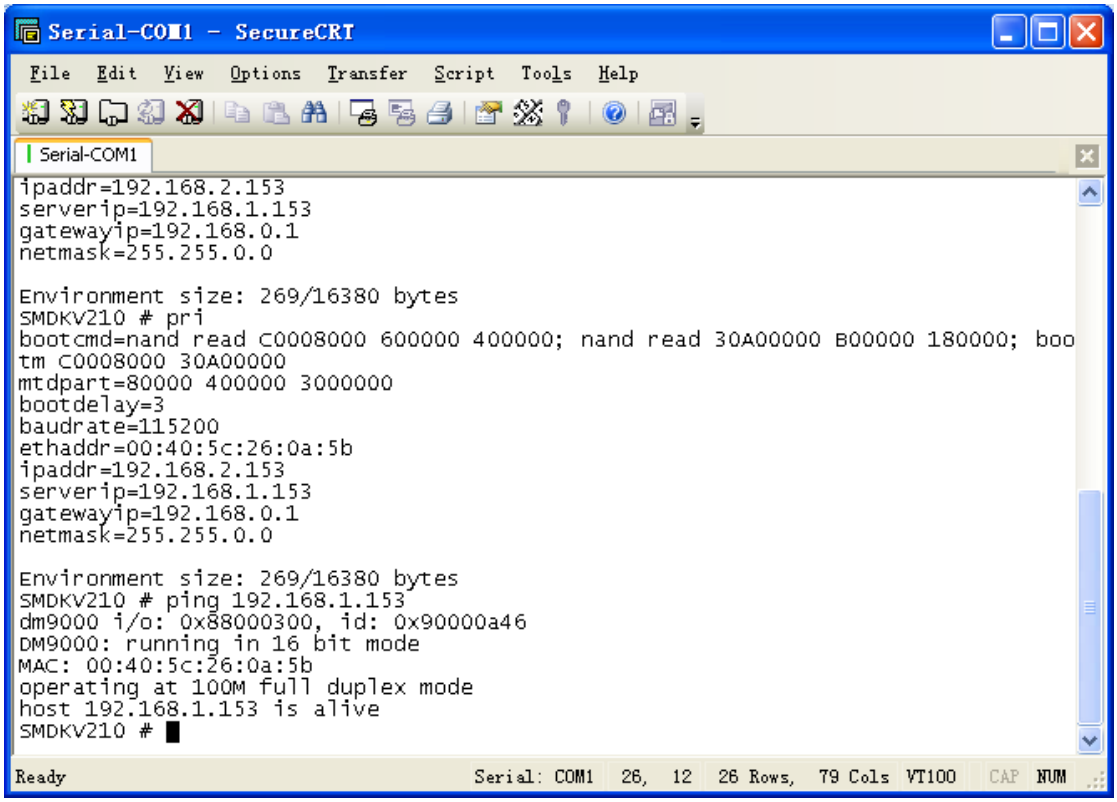


将开发板的 serverip 设置为 PC 机上的 IP 地址，将 ipaddr 设置为和 PC 机同一网段的 IP 地址：

```
setenv  serverip  192.168.1.153
setenv  ipaddr  192.168.2.153
saveenv
```

使用普通网线将网口连接到开发板和路由器，确保和 PC 机在同一网段路由，ping PC 机，测试硬件连接是否正常：

```
ping  192.168.1.153
```



如果提示 alive，表示硬件连接成功。

在 windows xp 下使用网口烧写映像时，需要使用虚拟机，或是 tftp 工具。这里以 tftp 工具为例，使用虚拟机，或是直接在 ubuntu 操作系统下操作时，方法类似。

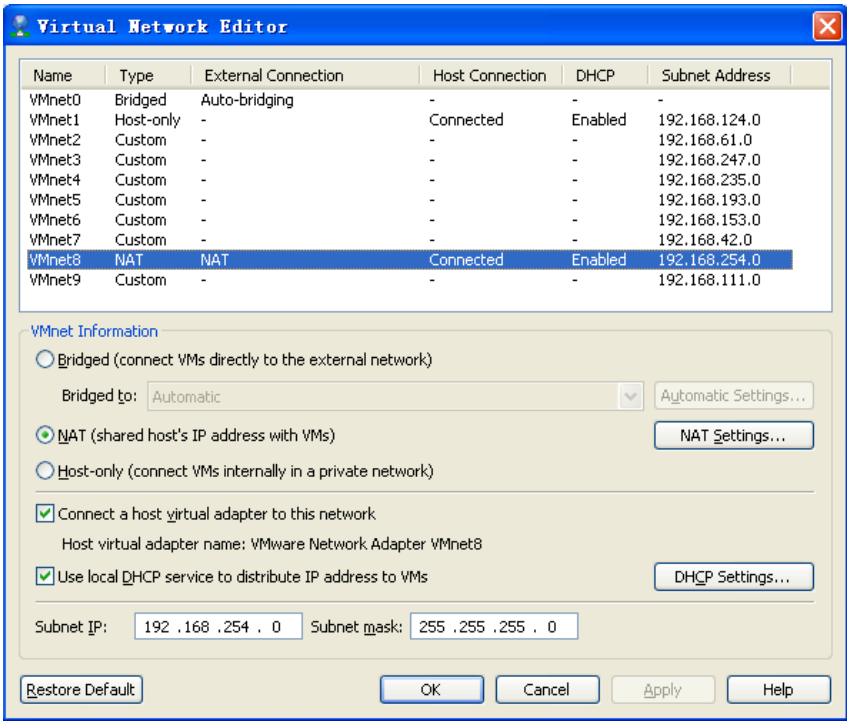
TFTPD32 是一个小巧，便捷的工具。我们可以通过它从服务器上下载 uboot，kernel 到目标板的 DRAM。当我们使用本机上的虚拟机时，就用不着 TFTPD32 了。

有时使用本机虚拟机后，再使用 TFTPD32 时，会出现如下错误：

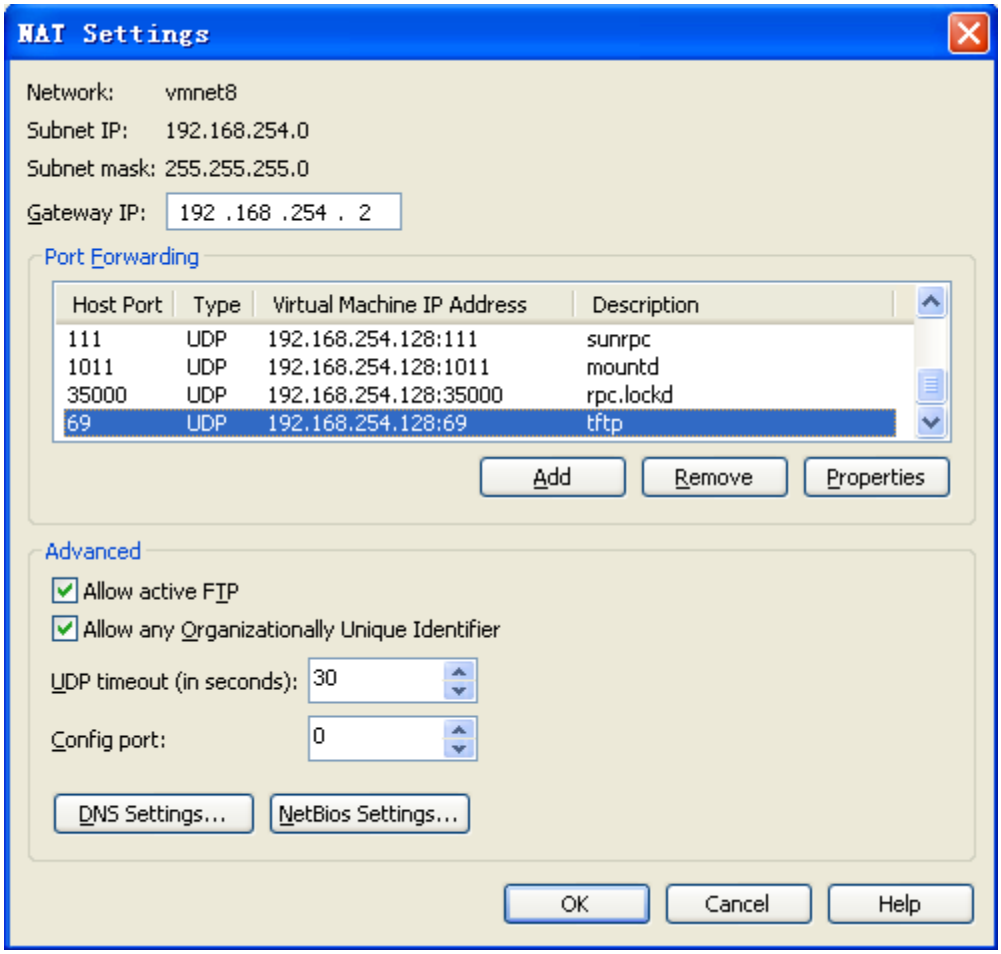
网上有人遇到这种情况，在 WINXP 的注册表中搜寻 TFTPD32，将所有该注册表全删掉，问题就解决了。但是有时候这种方法不一定行得通。上面提示可能已经有其他设备打开了 TFTP，当我们使用虚拟机时，如果使用了 NAT 上网方式，如果修改了里面的 TFTP 端口，这时虚拟机就占用了 TFTP，自然就会出现这种情况了。就算我们退出 VMWare，在后台仍

然还在运行。只有手动在任务管理器中关闭 VMWare 的使用到网络的进程，TFTPD32 才不会提示出错。但是杀掉这些进程后，我们想再回过头来用 VMWare 的 TFTP，就一定要重启计算机才行。

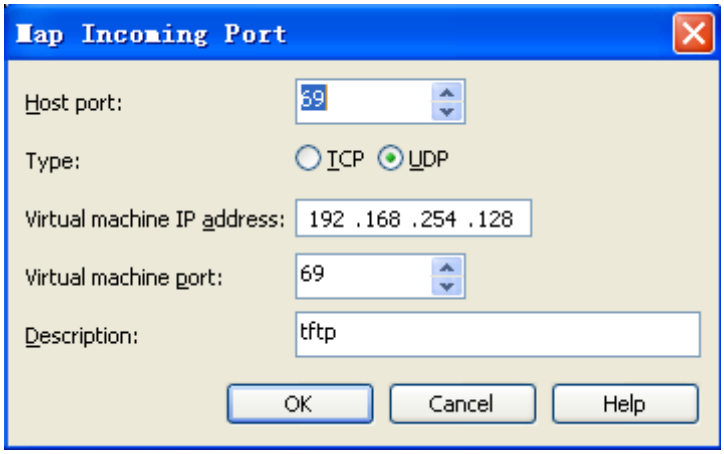
解决 TFTPD32 和虚拟机冲突的比较合适的办法是，在我们使用服务器时，将虚拟机的 69 端口设置为 UCP 模式，如果要回过头来使用虚拟机，将 69 端口设置为 UDP 模式即可。具体点击开始->程序->VMWare->Virtual Network Editor，在 Type 中选择 NAT，如下图：



点击 NAT Settings，选择端口号 69，如下图：



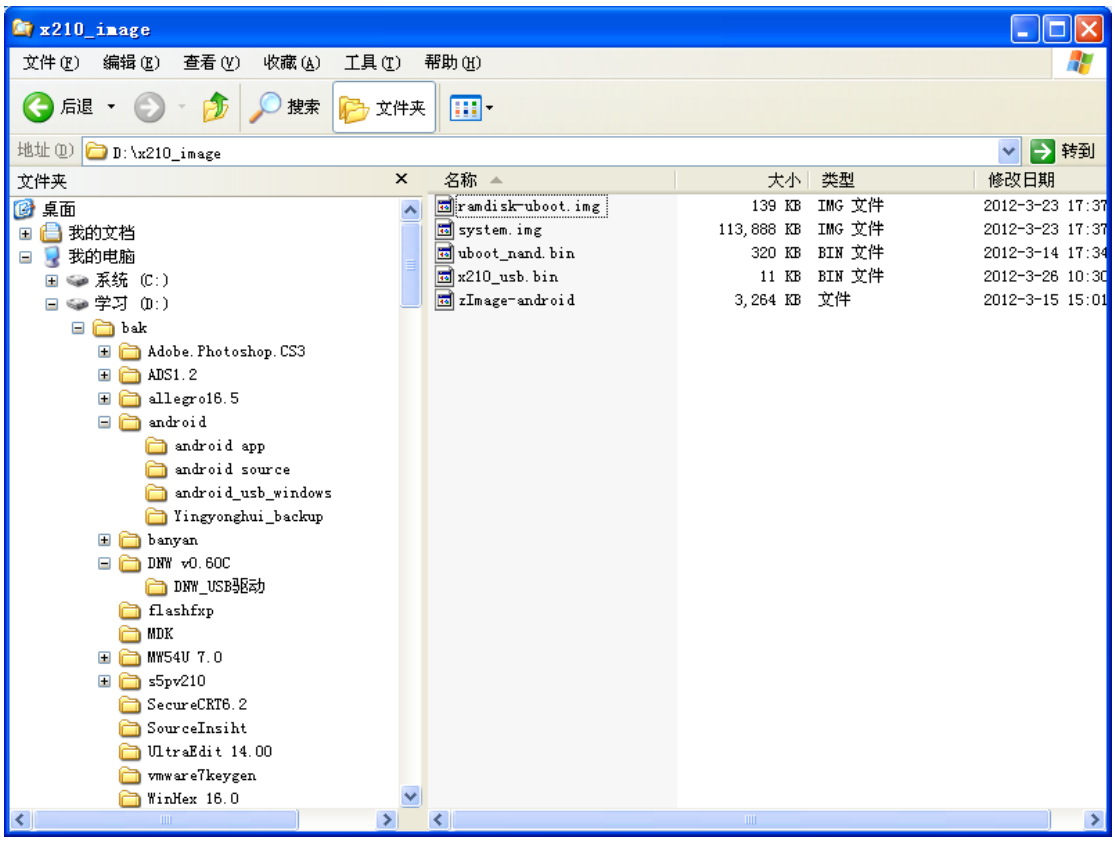
点击 Properties，将 TPYE 选择为 TCP，如下图：



这时再打开 TFTPD32 工具，上面的错误已经不再出现。如果需要回过头来使用虚拟机，重新选择 UDP 即可。

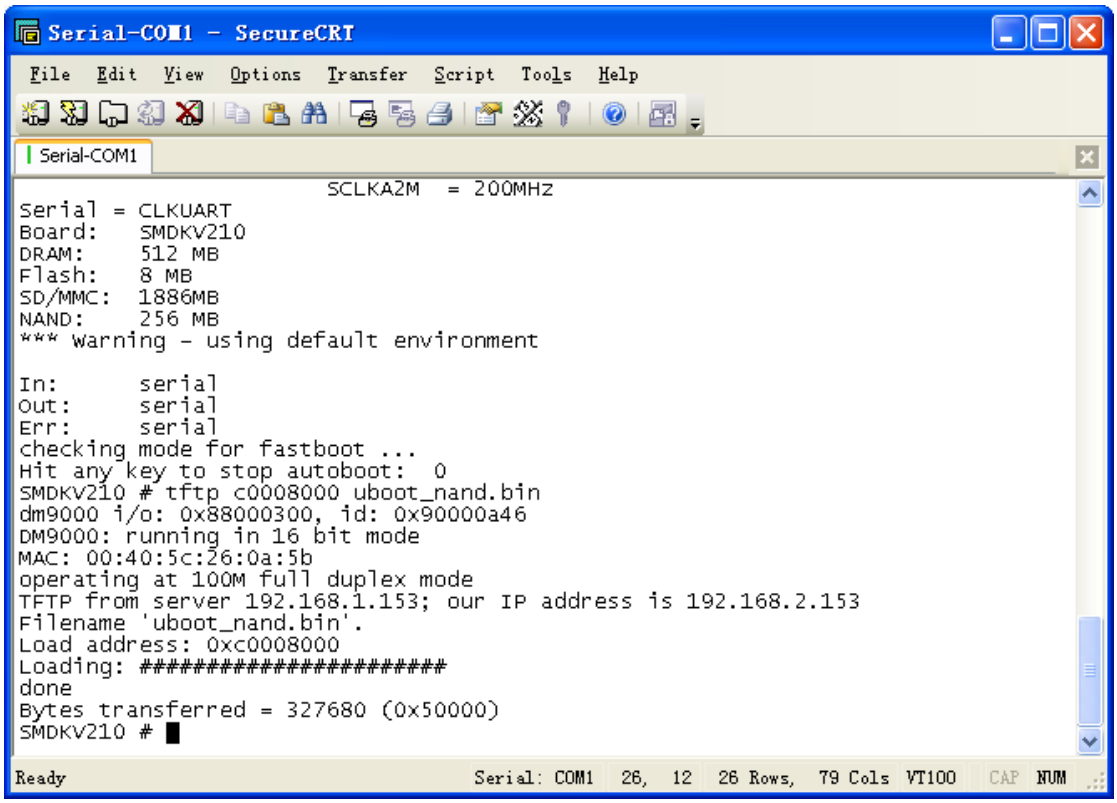
TFTPD32 正常打开后，点击 Settings，在 Base Directory 中选择映像放置的目录，如 D:\x210_image，点击 OK，Service interface 设置为 WINXP 的 IP 地址，TFTP 就可以正常使用了。

D:\x210_image 中文件内容如下：



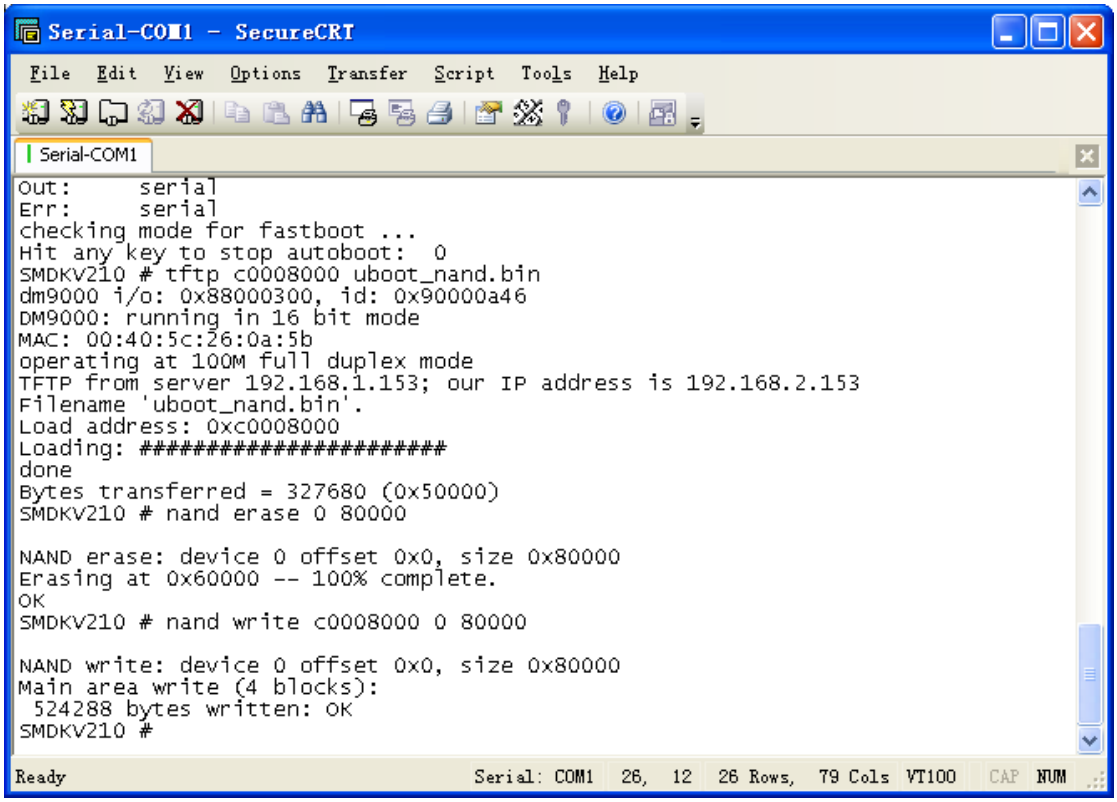
确保 tftpd32 工具在后台运行，执行如下指令下载 uboot 映像到 RAM：

```
tftp c0008000 uboot_nand.bin
```



使用 uboot 指令将下载的映像更新到 nand flash:

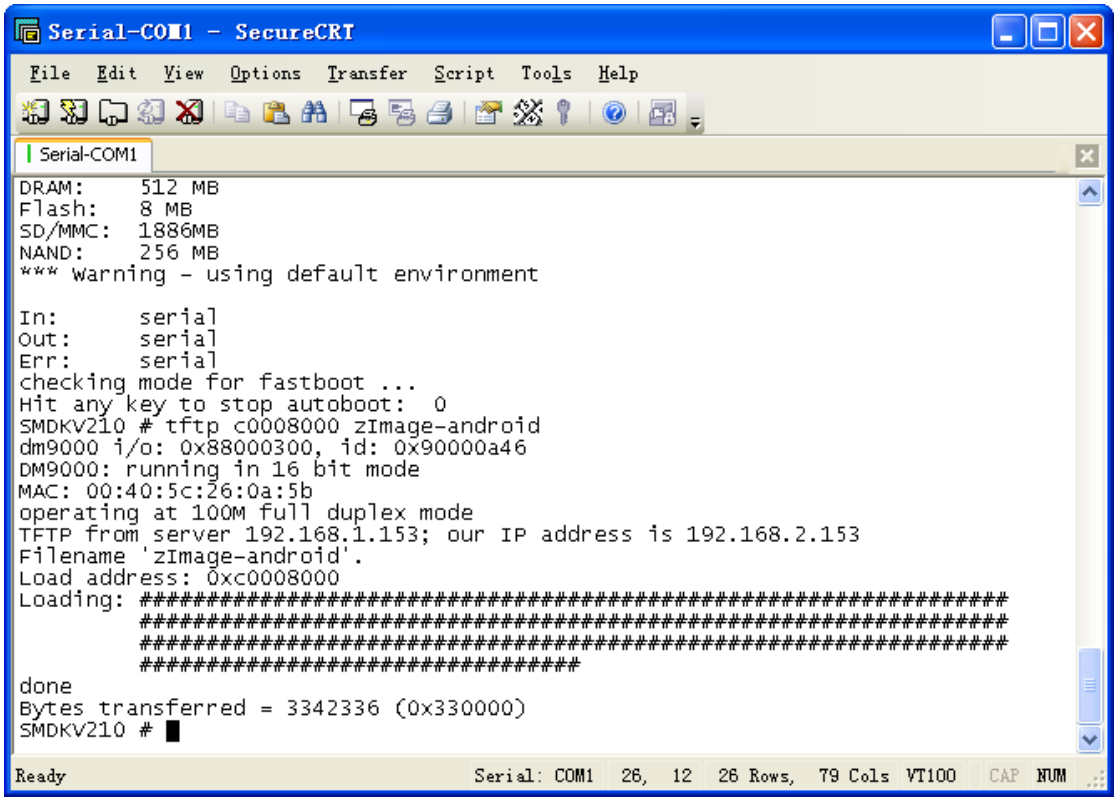
```
nand erase 0 80000
nand write c0008000 0 80000
```



6.3.4 在 windows 下使用网口更新 kernel

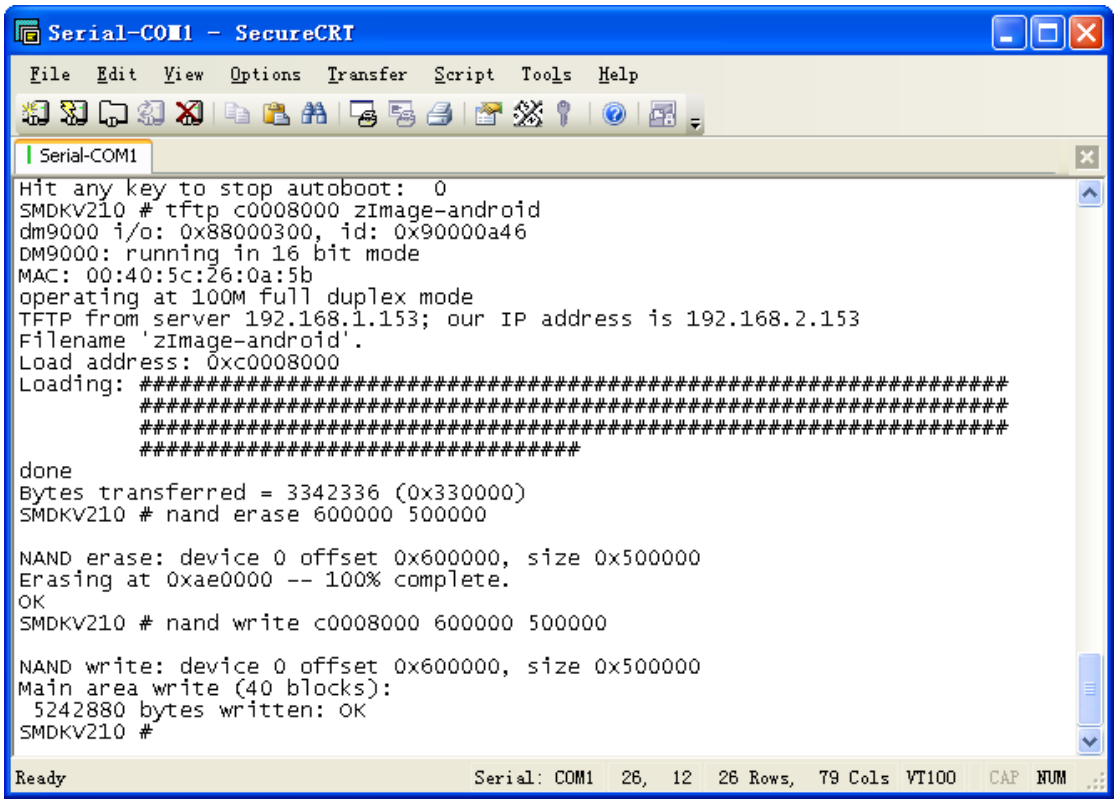
确保 tftpd32 工具在后台运行，执行如下指令下载内核映像到 RAM：

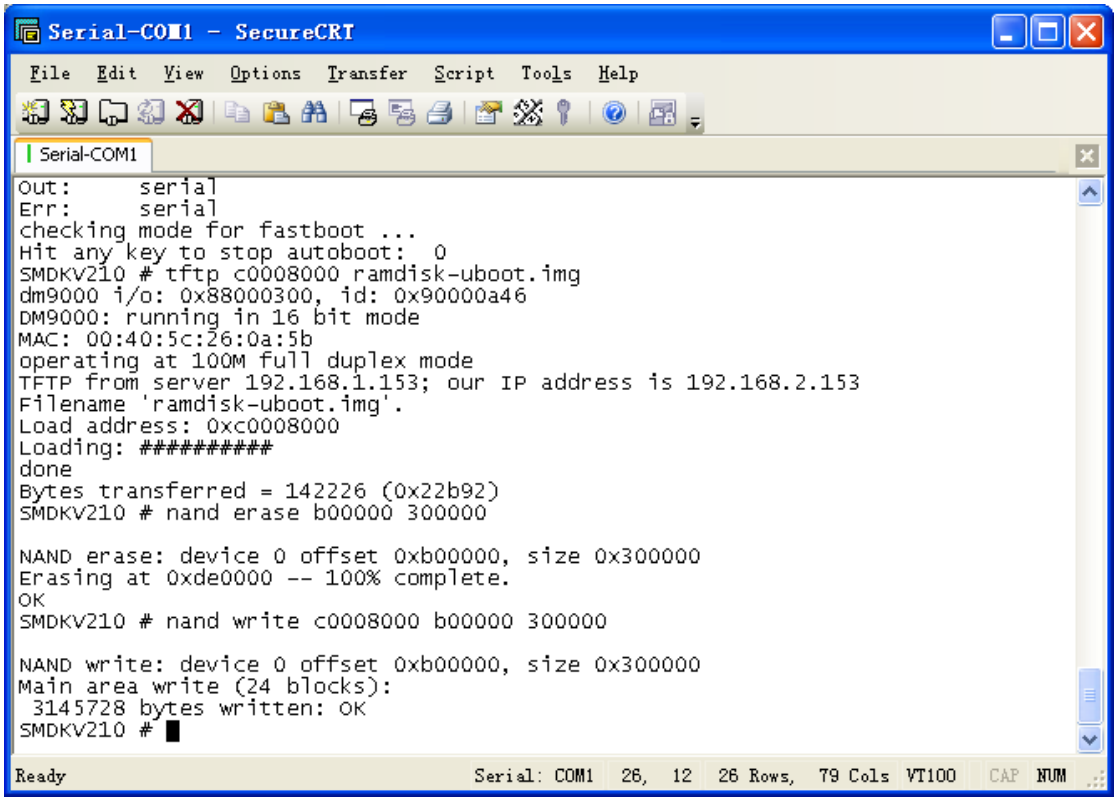
```
tftp c0008000 zImage
```



使用 uboot 指令将下载的映像更新到 nand flash:

```
nand erase 600000 500000
nand write c0008000 600000 500000
```

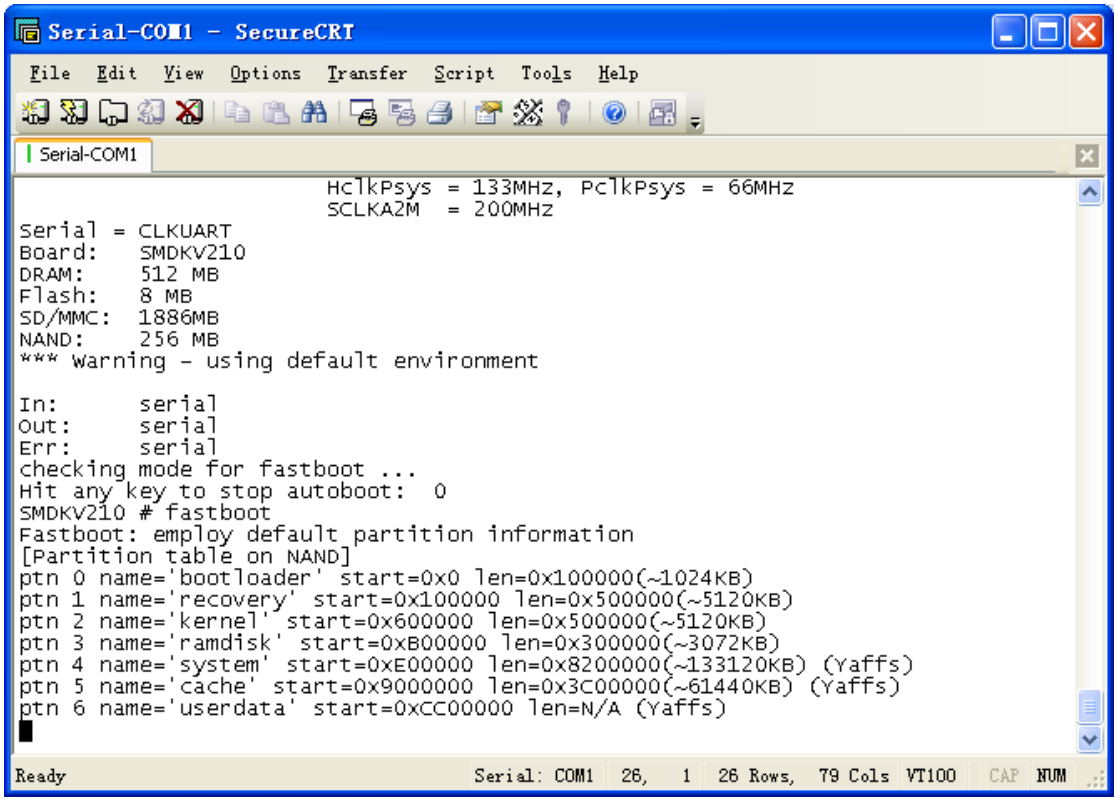




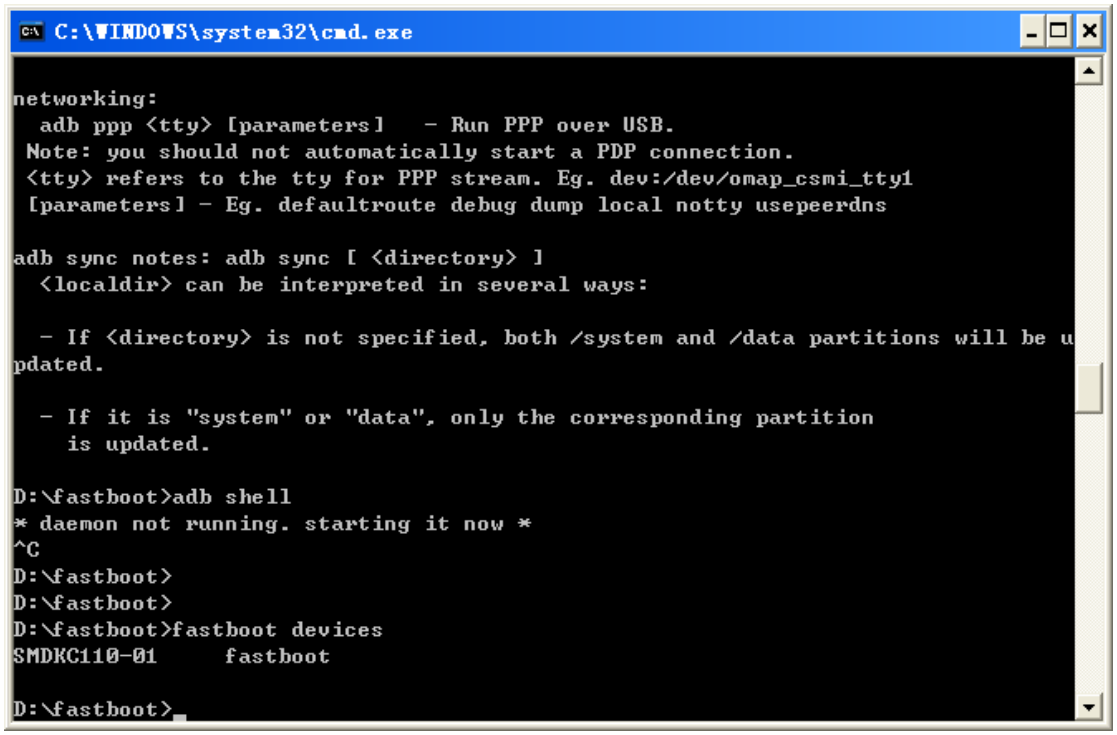
6.3.5 在 windows 下使用 fastboot 烧写映像文件

准备工作：将 windows 下的 fastboot 烧录工具 fastboot.exe 以及驱动文件 AdbWinApi.dll，AdbWinUsbApi.dll 放在 D:\fastboot 目录

在 uboot 命令行下，执行 fastboot 命令：



打开 windows 下的命令行终端，进入 D:\fastboot 目录，执行 fastboot devices 查询当前设备：

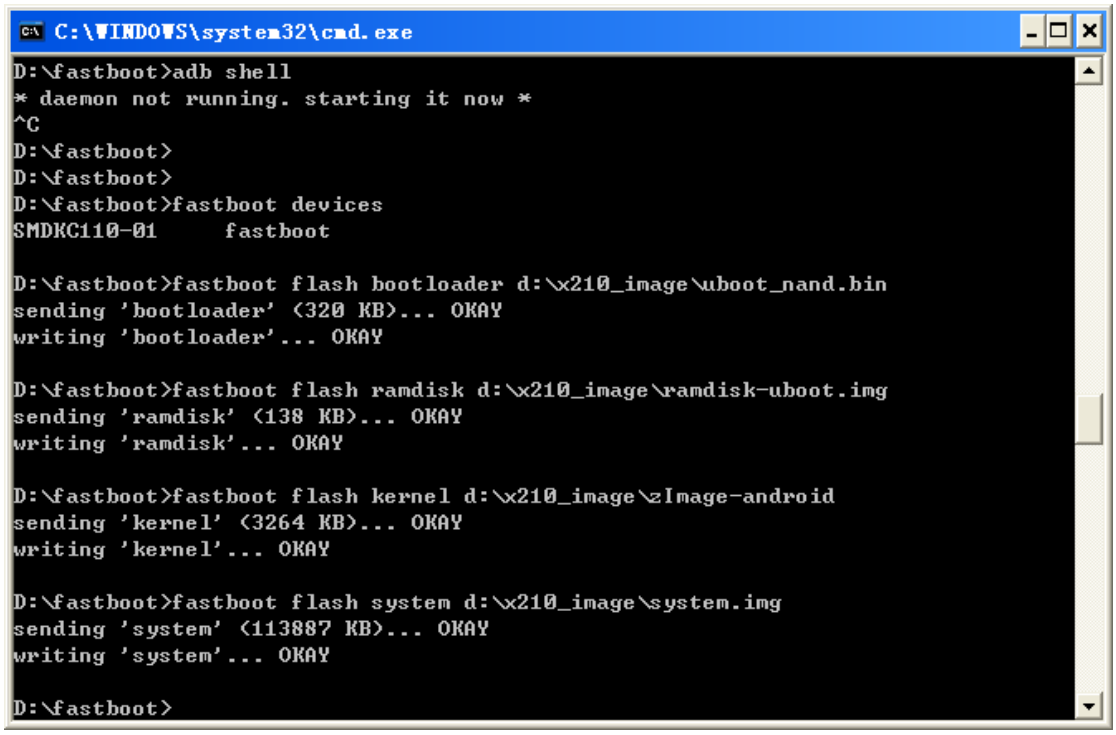


依次执行如下指令：

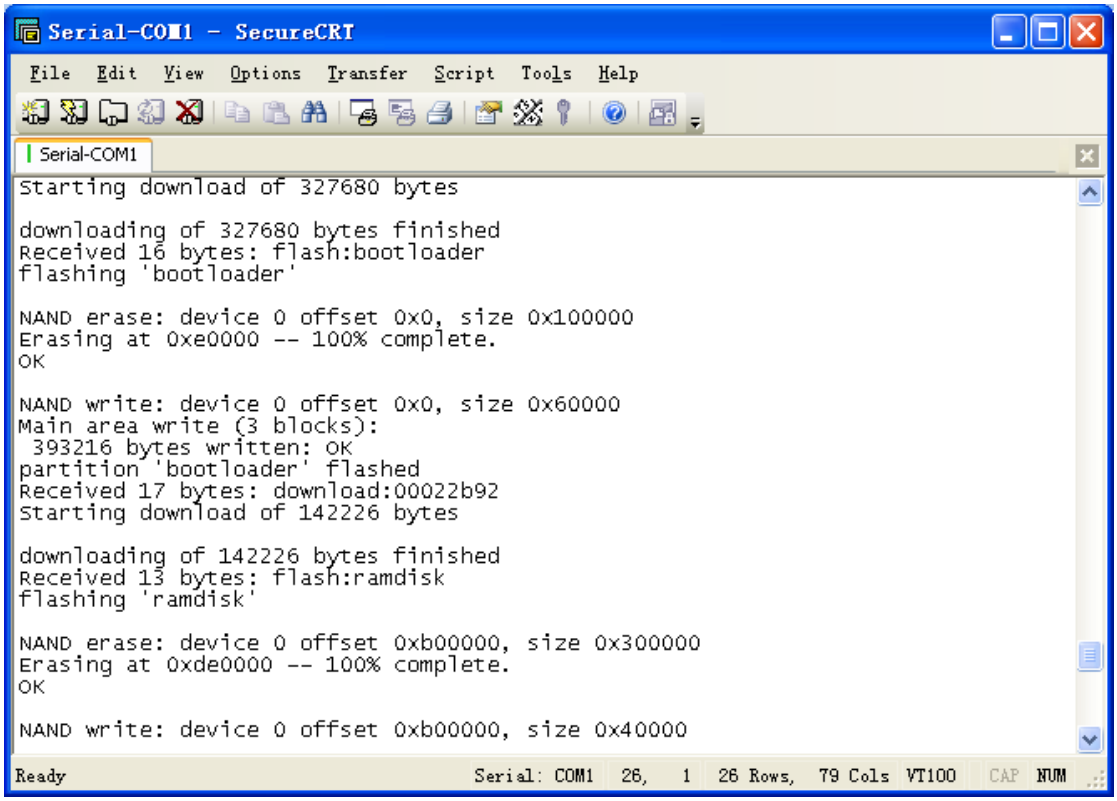
```
fastboot flash bootloader d:\x210_image\uboot.bin
fastboot flash kernel d:\x210_image\zImage
```

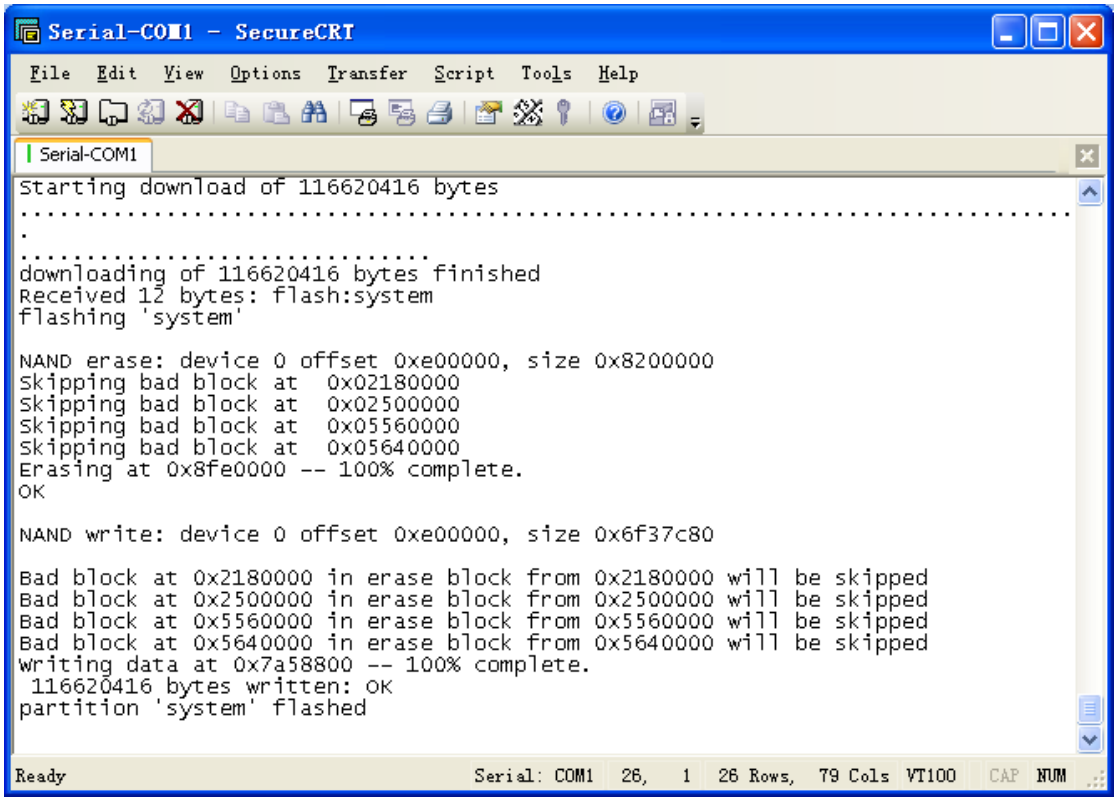
```
fastboot flash system d:\x210_image\x210.img
```

命令行终端烧录界面如下：



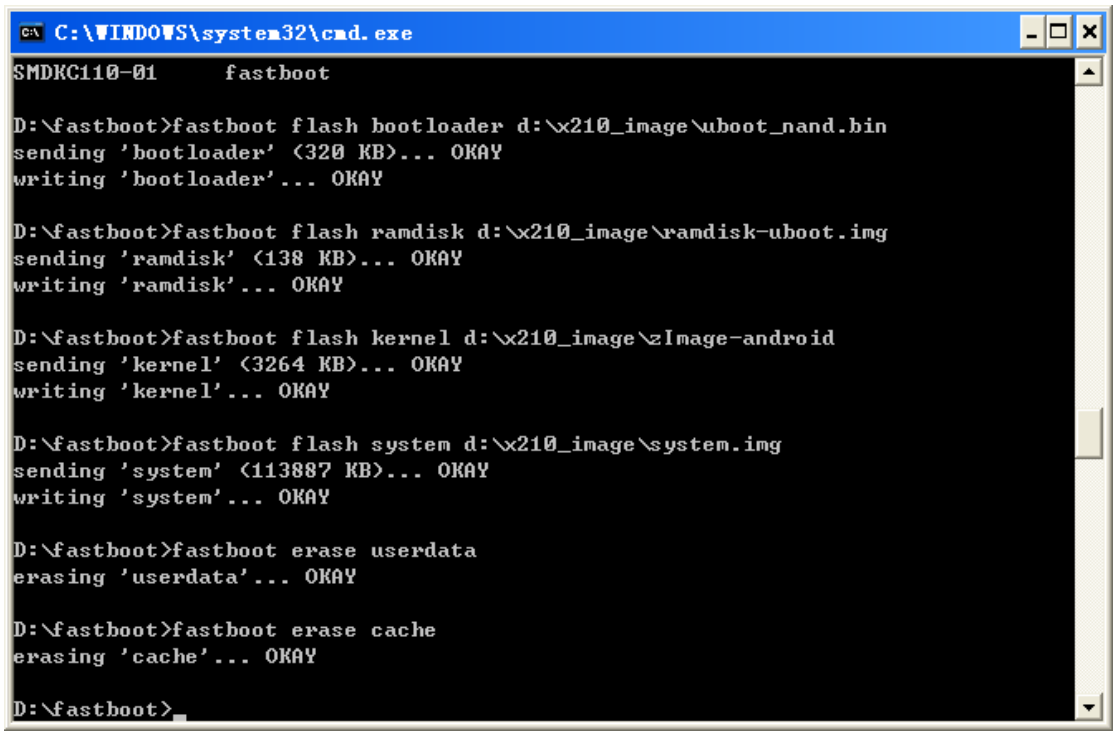
串口终端烧录信息如下：

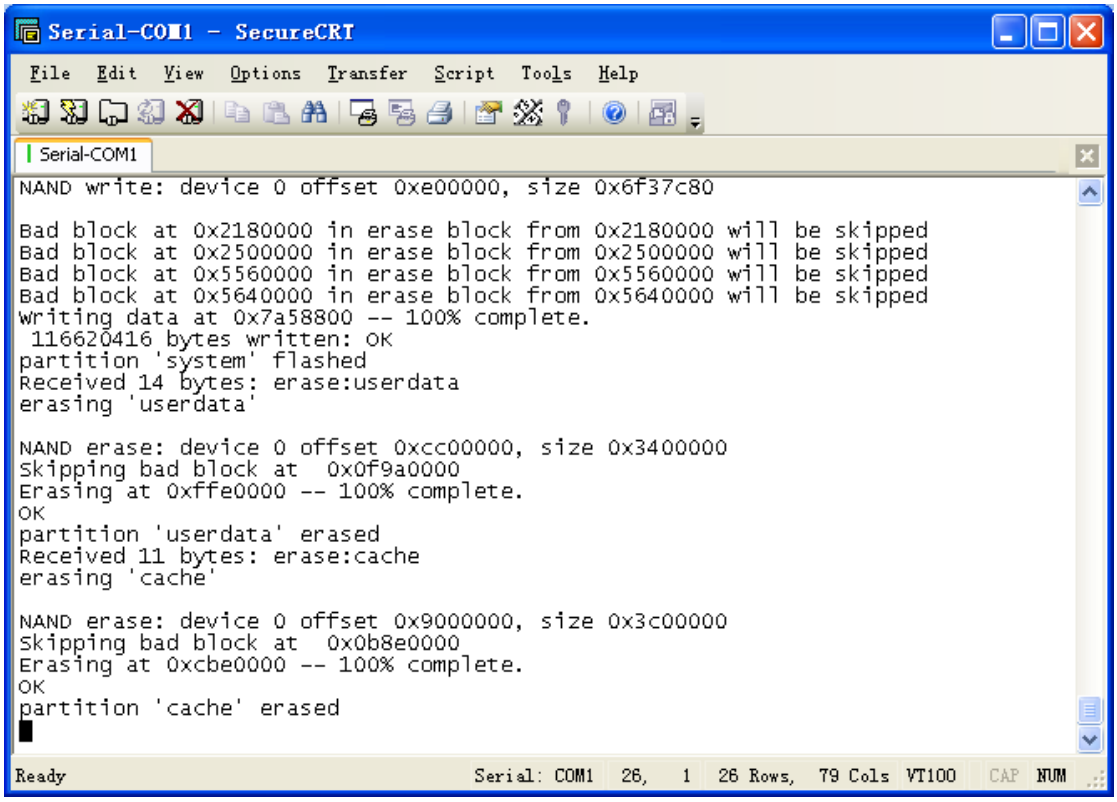




使用如下命令擦除用户数据和缓存区域：

```
fastboot erase userdata
fastboot erase cache
```





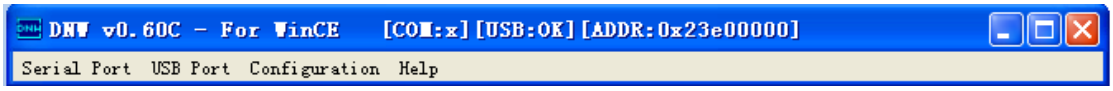
6.4 windows 下烧写映像文件到 inand

第一步：将拨码开关的 OM5 拨到 ON 位置，使用 USB 延长线连接 PC 机以及开发板的 miniUSB 接口；

第二步：使用串口延长线连接 PC 机以及开发板的 UART0；

第三步：打开 DNW，同时打开串口监控终端，注意，DNW 默认具有串口监控的功能，但是操作不是很方便，推荐使用 SerialCRT 工具；

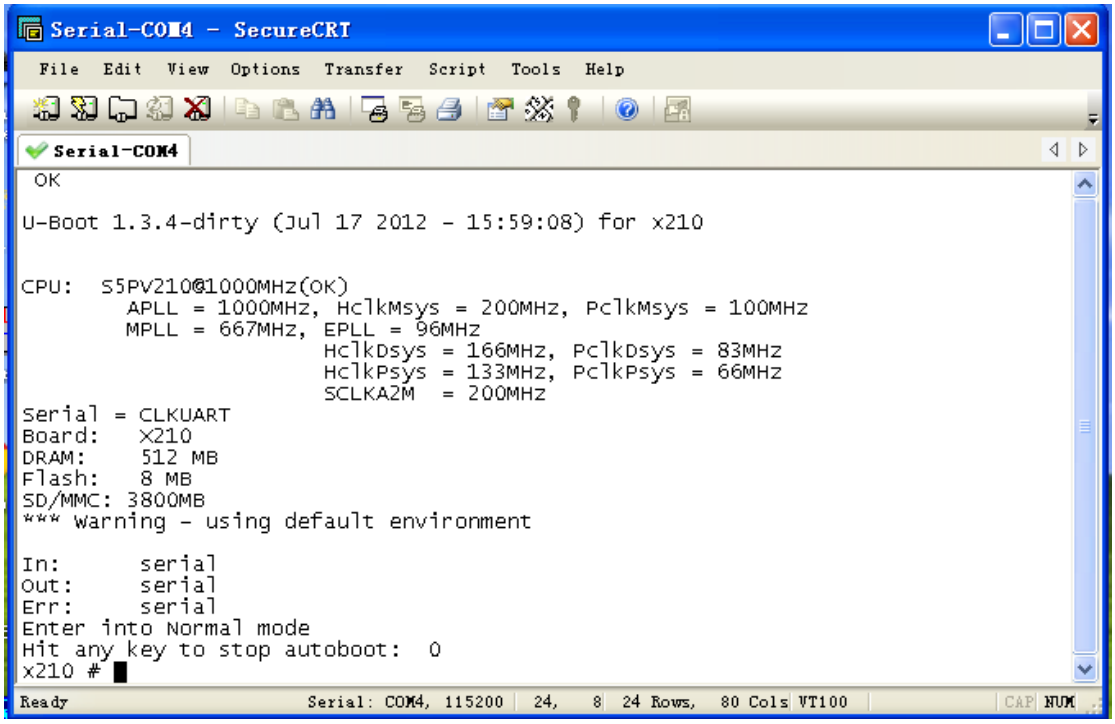
第四步：长按开发板的右下脚 POWER 键，这时，DNW 状态栏的 USB 一栏会显示 OK，如下图：



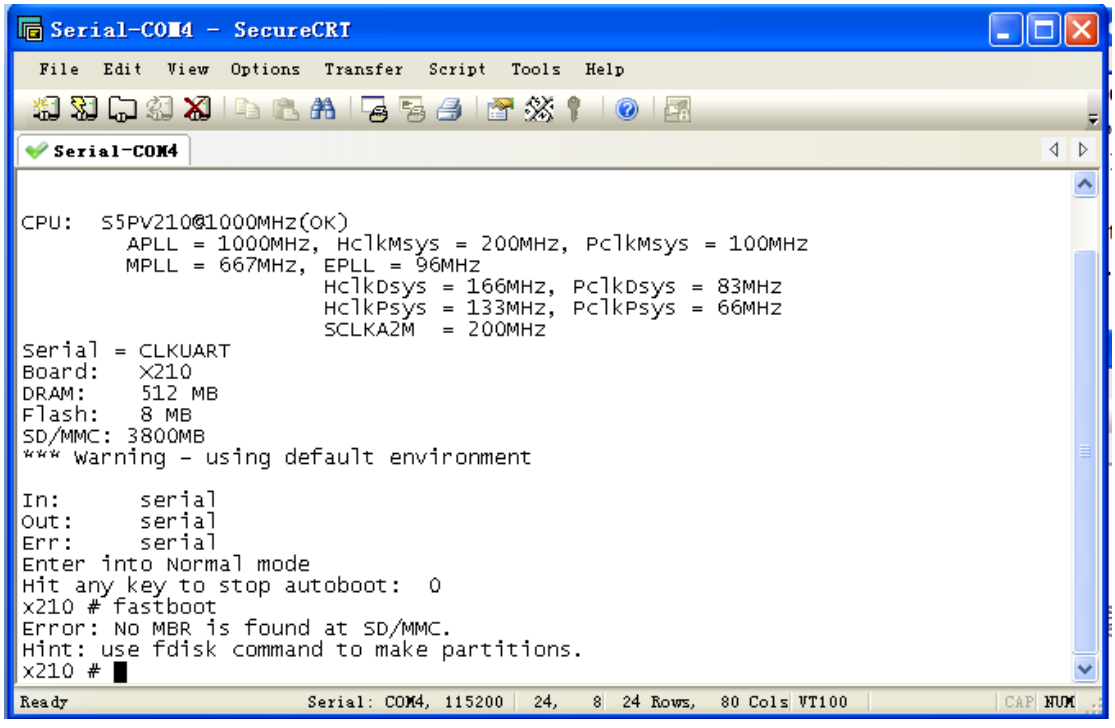
注意，在串口终端没有显示 uboot 打印信息时，不可松开 POWER 键！

第五步：将 DNW 的地址设置为 0xd0020010，点击 USB Port->Transmit，找到 x210_usb.bin 映像文件，双击下载，这时，DNW 的 USB 一栏会显示 USB:x 后，再显示 USB:OK，表明 USB 接口初始化成功；

第六步：将 DNW 的地址设置为 0x23e00000，点击 USB Port->Transmit，找到针对 inand 的 uboot.bin 映像文件，双击下载，这时，串口终端会显示 uboot 的打印信息，表明 uboot 已运行起来了，到此，便可以松开 POWER 键了。



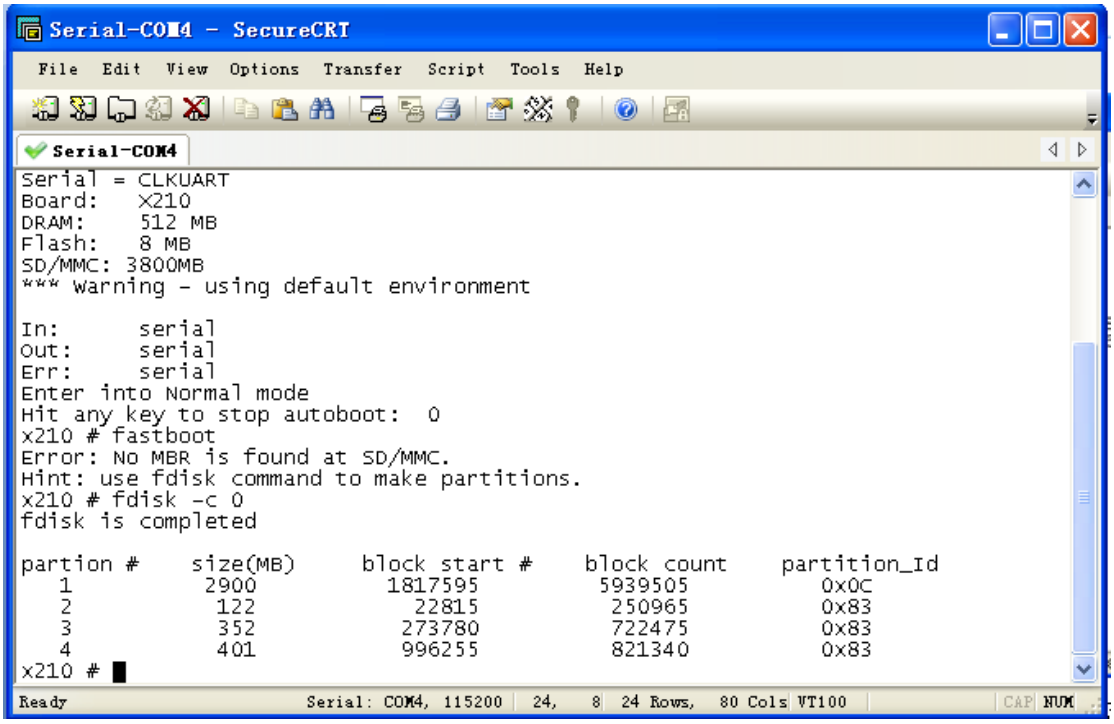
第七步：执行 fastboot 指令，回车，



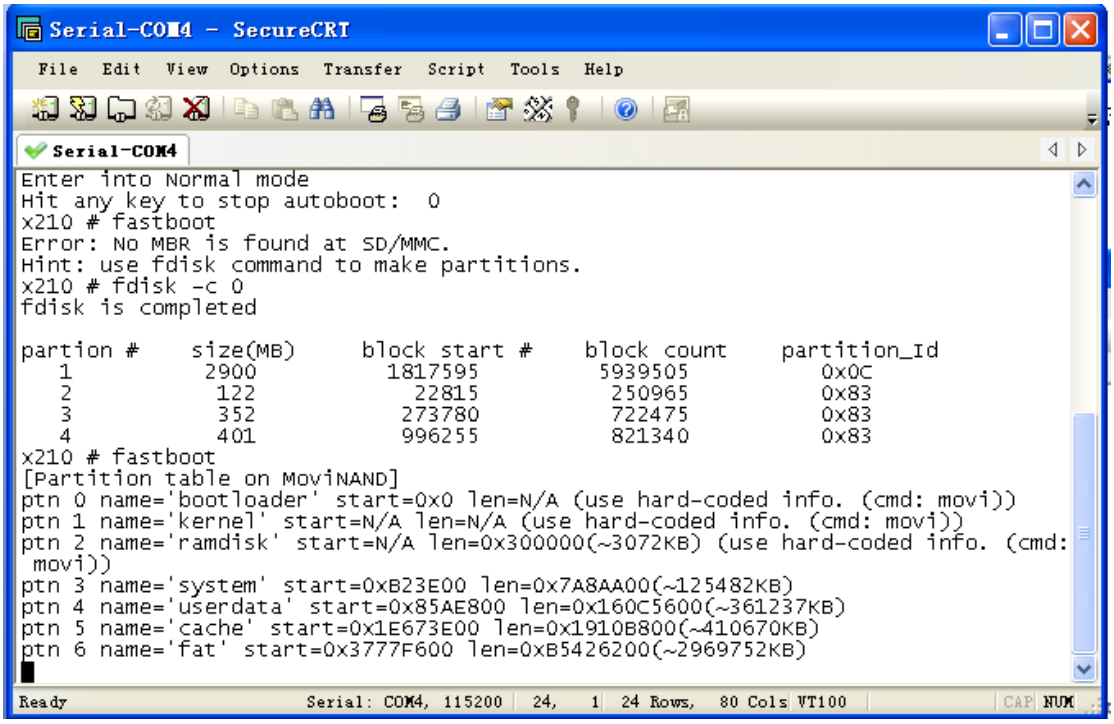
可以看到，上面会提示没有 MBR，这是因为，inand 芯片出厂时，是没有写 MBR 的，需要我们手动去分区。执行如下指令：

fdisk –c 0

分区信息如下：



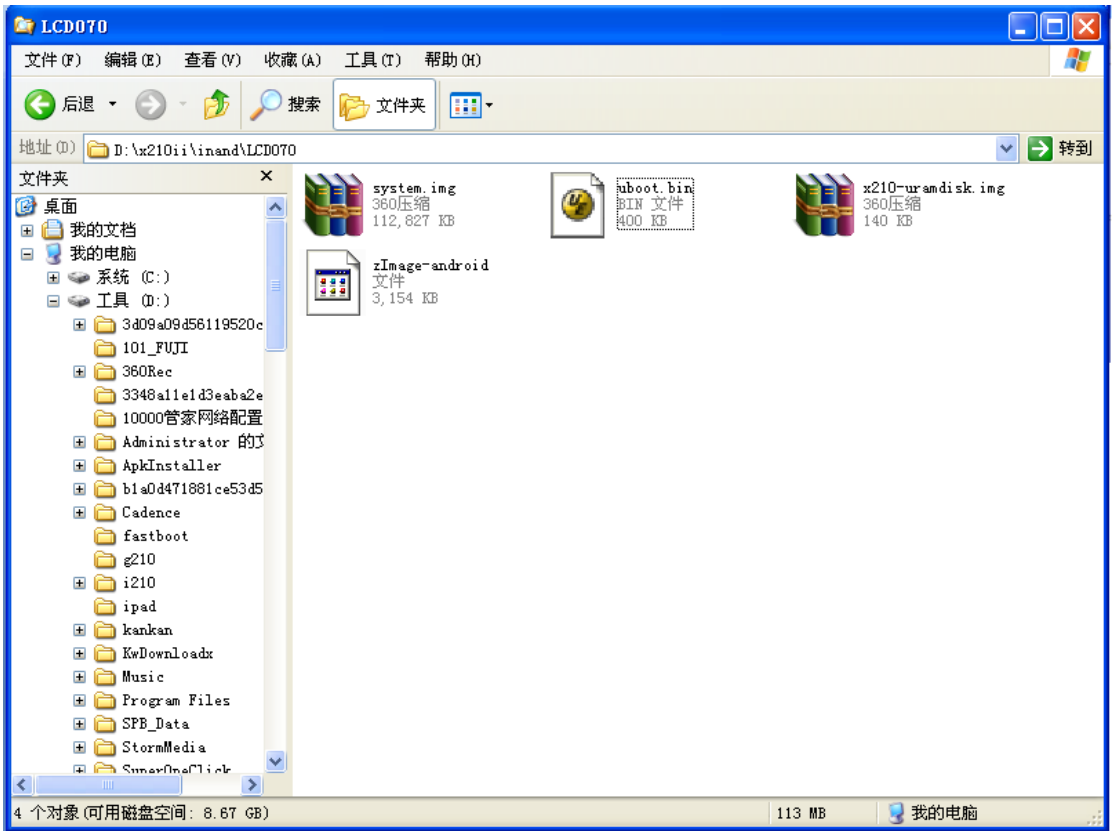
再执行 fastboot:



第八步：从光盘中将 fastboot 目录拷贝到 PC 机的某目录，如 D 盘根目录，确保存在以下文件：

adb.exe
AdbWinApi.dll
AdbWinUsbApi.dll
fastboot.exe

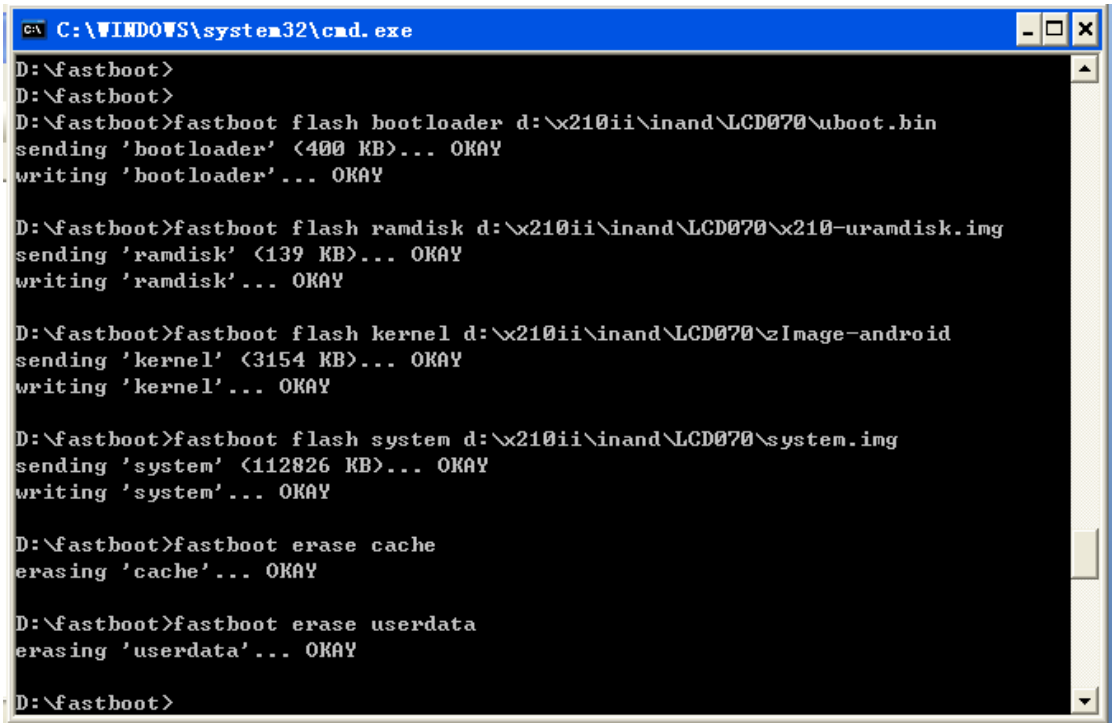
第九步：将编译生成的映像文件拷贝到 PC 机的某目录，如：



第十步：打开 PC 机的命令行界面，进入 fastboot 目录，依次执行如下指令：

```
fastboot  flash  bootloader  d:\x210v3s\inand\LCD070\uboot.bin
fastboot  flash  kernel  d:\x210v3s\inand\LCD070\zImage
fastboot  flash  system  d:\x210v3s\inand\LCD070\x210.img
fastboot  -w
```

命令行执行成功界面如下：



第十一步：将拨码开关的 OM[5:0]设置为 001101，重启开发板，这时系统就已经从 inand 运行起来了。

6.5 ubuntu 下烧写映像文件到 inand

在 ubuntu 下烧写映像文件到 inand 时，无法设置从 USB 启动了，因为在 ubuntu 下没有 DNW 软件，也就无法将程序直接下载到内存运行了。在 inand 中没有映像文件时，我们可以将针对 inand 的 uboot 映像烧写到 SD 卡，再从开发板的卡槽 2 启动，然后执行 fastboot 命令更新到 inand。

第一步：将针对 inand 的 uboot 映像烧写到 SD 卡，详见 6.1 章节。

第二步：将 SD 卡插到开发板的右侧卡槽，即 SD2 通道；

第三步：长按开机键开机，注意一定要确保 inand 中没有映像文件，否则无法从 SD2 启动。

第四步：在串口终端 ，在 3 秒倒计时内按空格键，进入 uboot 命令行，执行如下指令：

```
fdisk  -c  0
```

将会给 inand 分区；

第五步：使用如下指令更新映像：

```
fastboot  flash  bootloader  uboot.bin
fastboot  flash  kernel  zImage
fastboot  flash  system  x210.img
```

第7章 android 开发指南

7.1 触摸屏校正

对于电容屏驱动，触摸屏不需要校正，程序已固化在驱动中，对于电阻屏驱动，有一个校屏的测试程序，但是事实上，校正的值也已经固定在驱动中，如需细调校正参数，需手动获取校正参数后，固化到驱动中，具体校正方法需消化驱动程序后再进行校正。

7.2 播放 mp3

7.2.1 android 命令行播放 mp3

在 android 的在命令行下，可以使用强大的 am 指令做很多事情。在 android 终端输入 am，正常情况下会有如下提示信息：

```
# am
usage: am [subcommand] [options]

       start an Activity: am start [-D] [-W] <INTENT>
             -D: enable debugging
             -W: wait for launch to complete

       start a Service: am startservice <INTENT>

       send a broadcast Intent: am broadcast <INTENT>

       start an Instrumentation: am instrument [flags] <COMPONENT>
             -r: print raw results (otherwise decode REPORT_KEY_STREAMRESULT)
             -e <NAME><VALUE>: set argument <NAME> to <VALUE>
             -p <FILE>: write profiling data to <FILE>
             -w: wait for instrumentation to finish before returning

       start profiling: am profile <PROCESS> start <FILE>
       stop profiling: am profile <PROCESS> stop

<INTENT> specifications include these flags:
             [-a <ACTION>] [-d <DATA_URI>] [-t <MIME_TYPE>]
             [-c <CATEGORY> [-c <CATEGORY>] ...]
             [-e|--es <EXTRA_KEY><EXTRA_STRING_VALUE> ...]
             [--esn <EXTRA_KEY> ...]
             [--ez <EXTRA_KEY><EXTRA_BOOLEAN_VALUE> ...]
             [-e|--ei <EXTRA_KEY><EXTRA_INT_VALUE> ...]
             [-n <COMPONENT>] [-f <FLAGS>]
             [--grant-read-uri-permission] [--grant-write-uri-permission]
             [--debug-log-resolution]
             [--activity-brought-to-front] [--activity-clear-top]
```

```
        [--activity-clear-when-task-reset] [--activity-exclude-from-recents]
        [--activity-launched-from-history] [--activity-multiple-task]
        [--activity-no-animation] [--activity-no-history]
        [--activity-no-user-action] [--activity-previous-is-top]
        [--activity-reorder-to-front] [--activity-reset-task-if-needed]
        [--activity-single-top]
        [--receiver-registered-only] [--receiver-replace-pending]
        [<URI>]
```

启动的方法为：

```
# am start -n  包(package)名/包名.活动(activity)名称
```

启动的方法可以从每个应用的 AndroidManifest.xml 的文件中得到，以计算器（calculator）为例，

```
<?xml version="1.0" encoding=""?>

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

package="com.android.calculator2">

<applicationandroid:label="@string/app_name"android:icon="@drawable/icon">

<activity android:name="Calculator"

android:theme="@android:style/Theme.Black">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<categoryandroid:name="android.intent.category.LAUNCHER"/>

</intent-filter>

</activity>

</application>

</manifest>
```

由此计算器（calculator）的启动方法为：

```
# am start-ncom.android.calculator2/com.android.calculator2.Calculator
```

Music 的启动方法为：

```
# am start-ncom.android.music/com.android.music.MusicBrowserActivity
```

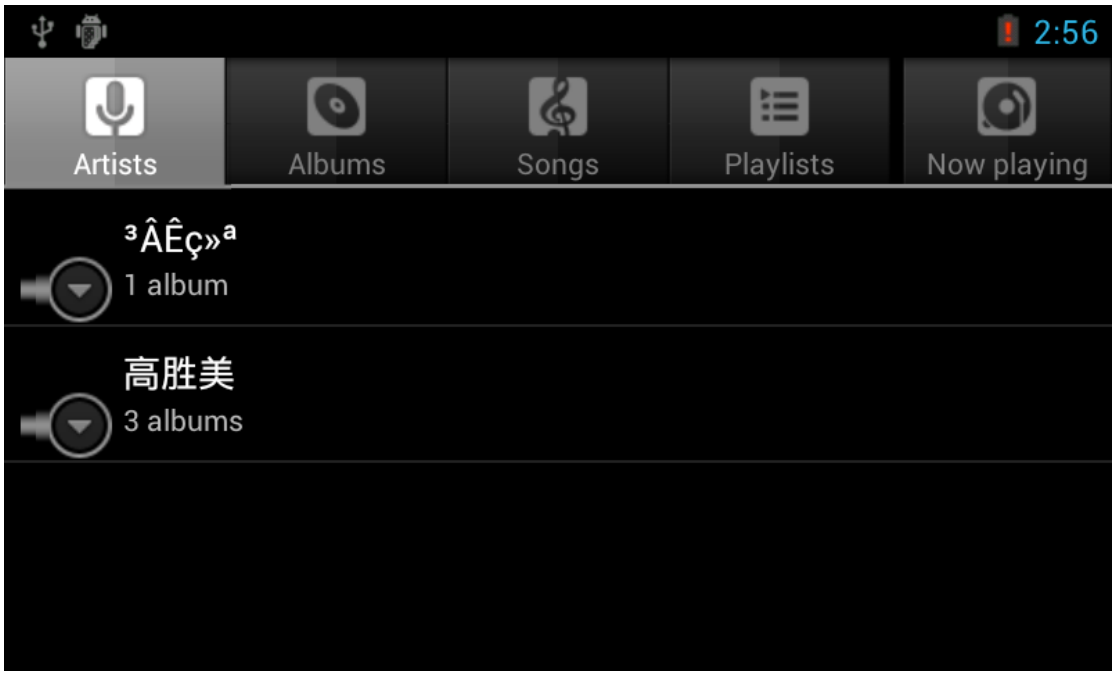
这时，屏幕上会有 music 的播放列表，但是并没有播放。如果需要播放，得执行下面的

指令：

```
am start -n com.android.music/com.android.music.MediaPlaybackActivity -d /sdcard/liangliangxianwang.mp3
```

7.2.2 使用 android 默认音频播放器

确保外置的 SD 卡中在在 mp3 文件，需要注意的是，SD 卡一定要插在右侧卡槽内。点击音乐，播放器会自动识别音频文件，如下图：

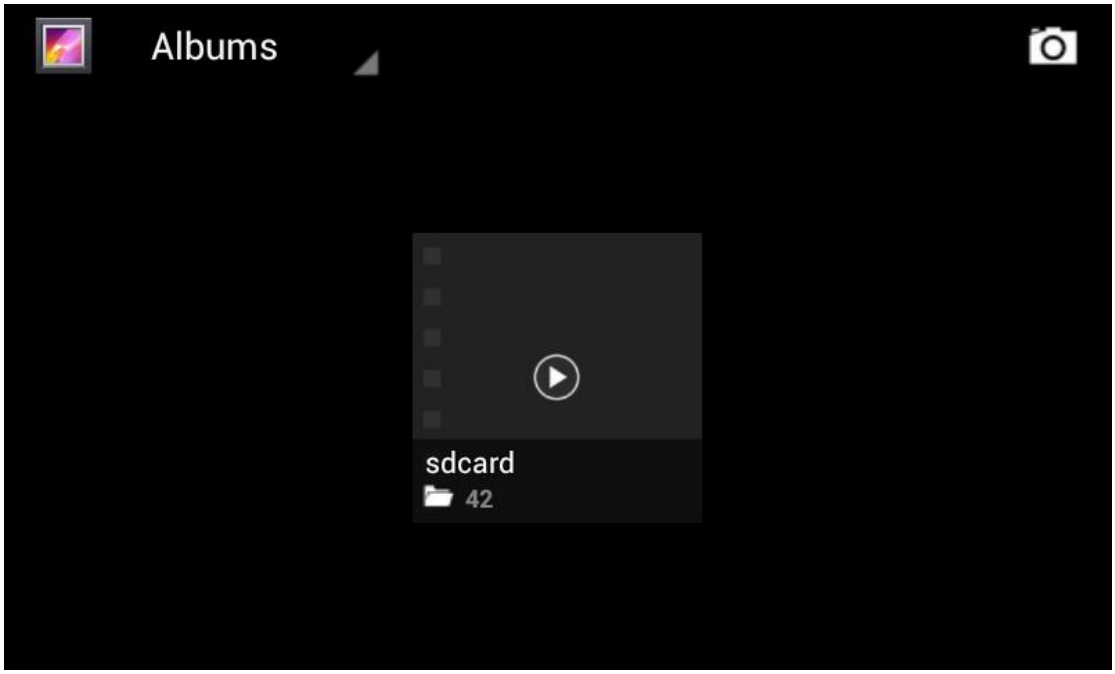


点击相应的音频文件即可播放。播放时界面如下：

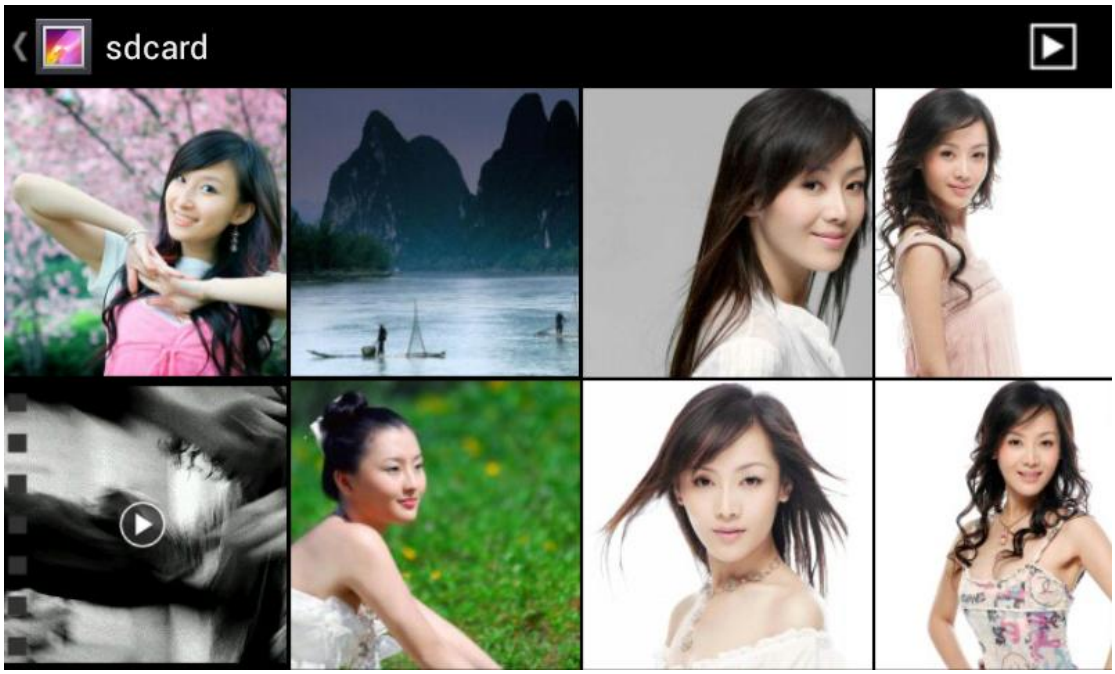
7.3 播放视频

android 自带视频处理功能，在 android 应用界面显示为图库。点击图库按钮，会在外置 SD

卡中自动寻找能够识别的视频和图片文件，如下图：

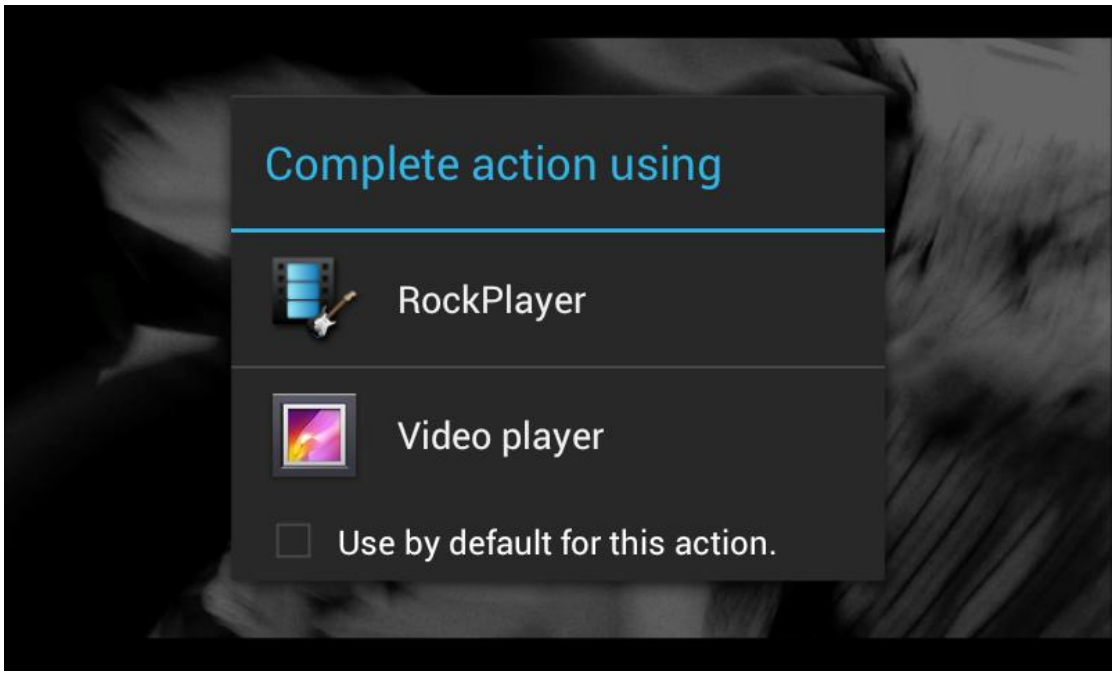


点击中间的按钮：



上图中，带有播放符号的即为视频文件，不带的为图片文件。点击带播放符号的文件：

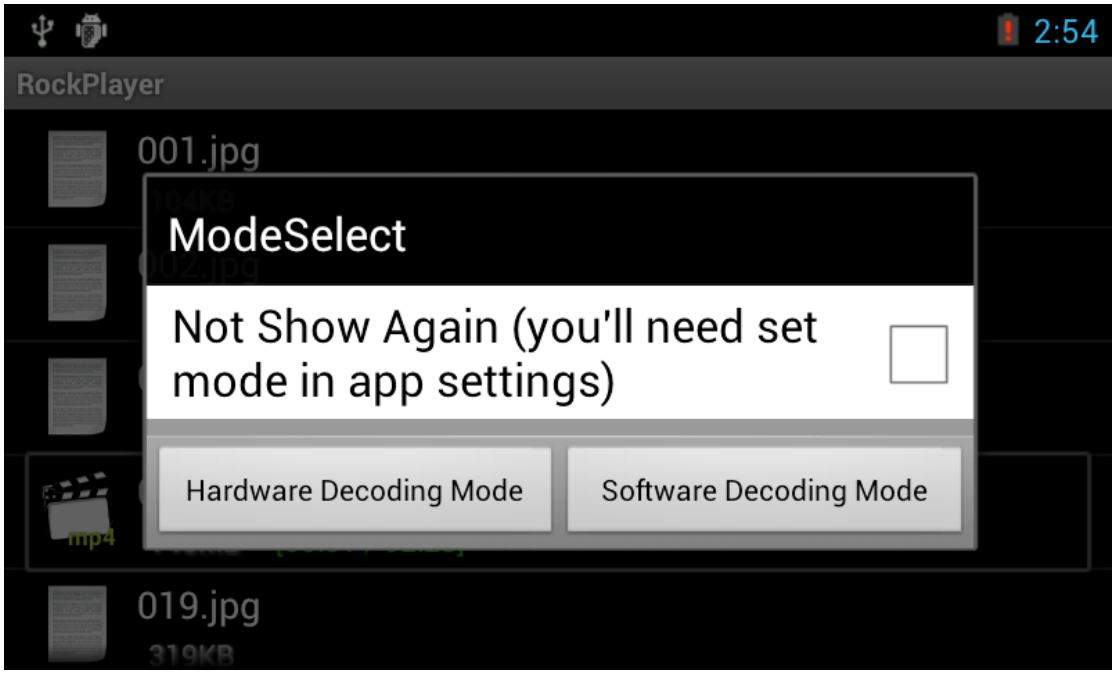
再点击播放按钮：



这时系统提示选择使用什么播放器播放，因为默认出厂时，我们已经将 RockPlayer 装到 x210 开发板上了。任意选择一个播放器即可播放。

如果遇到图库无法支持的视频文件，可以采用第三方播放器如 RockPlayer 进行播放，如网络上最为流行的 rmvb 和 rm 文件。这时，机器俨然成为了一个具有支持 rm/rmvb 等格式视频的超强 mp4 了。如对屏幕尺寸有更高要求，可以使用后面的 VGA 或 HDMI 方式，直接将视频文件显示到显示器或电视机上。

使用 RockPlayer 播放视频时，会弹出一个硬解和软解的对话框，如果属于 210 硬解码的视频文件，选择硬解模式，否则选择软解模式。如播放 rm/rmvb 文件，选择软解模式才能播放，如下图所示：



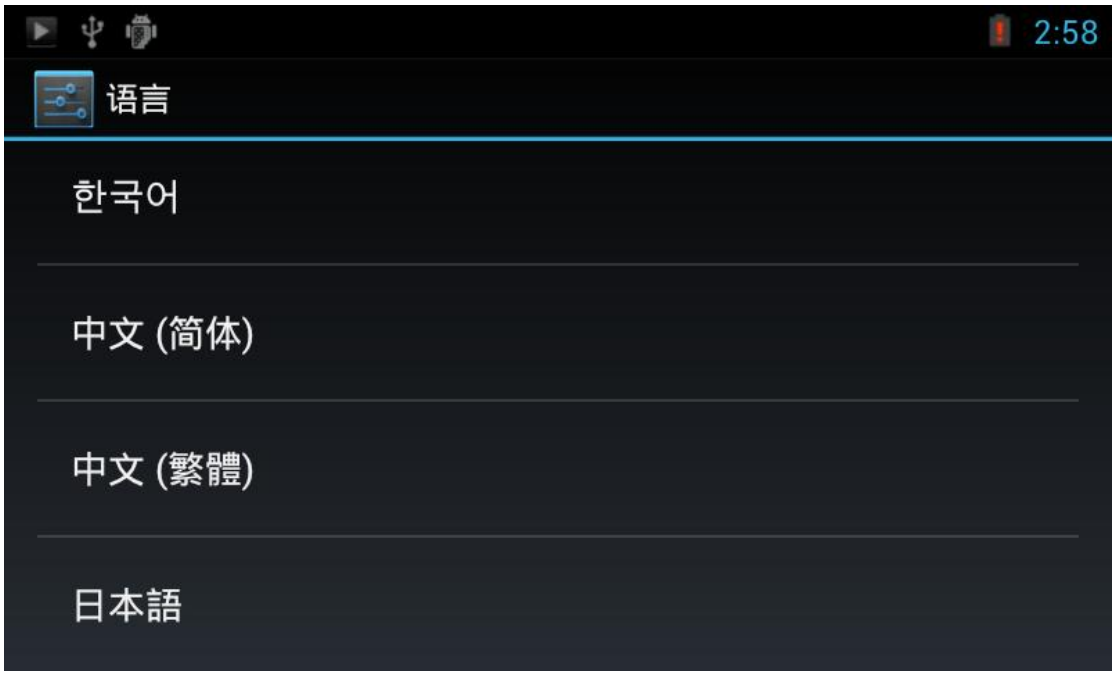
7.4 图片浏览

浏览图片时，同样使用上面的图库浏览。点击图库图标，点击要浏览的图片即可浏览。

滑动可以浏览下一张图片，如下图所示：

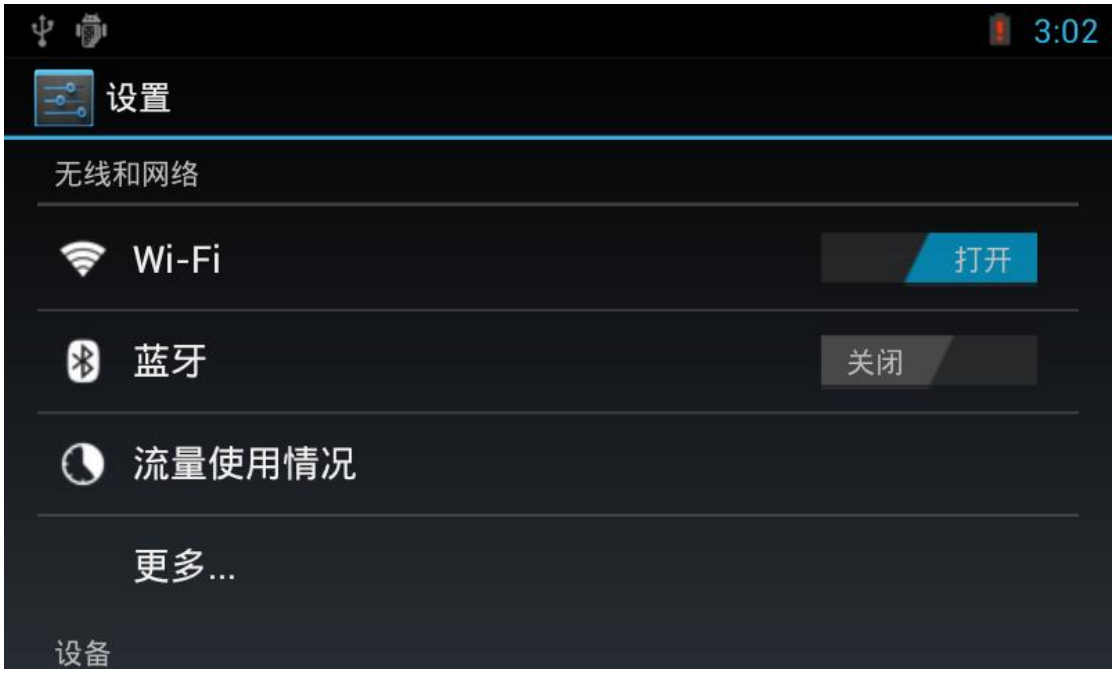
7.5 语言设置

点击设置中的语言和键盘一栏，再点击选择语言，会弹出多种语言，选择需要的语言即可，如下图：



7.6 使用 WIFI 上网

将 USB WIFI 插到 USB 的任意一个 HOST 接口，点击设置，在 Wi-Fi 一栏的方框中有个关闭按钮，将他拨到右边，即打开状态，如下图：



再点击 Wi-Fi 一栏，进入 wifi 界面，找到属于自己的无线网络信号并点击，输入密码：

再点击连接，成功连接后界面如下：

7.7 使用蓝牙传输数据

x210v3s android4.0.4 支持串行接口蓝牙模块，注意，使用蓝牙时，请务必先将蓝牙模块插到 x210v3s 开发板的 J13 接口。

点击设置，可以看到有个蓝牙选项，点击打开，就可以进行蓝牙传输了。

7.8 使用 USB 鼠标

将 USB 鼠标或者 USB 无线鼠标接到 USB HOST 接口，即可使用鼠标操作 android 界面了。

7.9 使用 USB 键盘

将 USB 键盘接到 USB HOST 接口，即可使用键盘操作 android 了。

7.10 APK 应用安装

待续

7.10.1 使用 SD 卡安装

待续

7.10.2 使用 ApkInstaller 安装

待续

7.10.3 使用 adb 工具安装

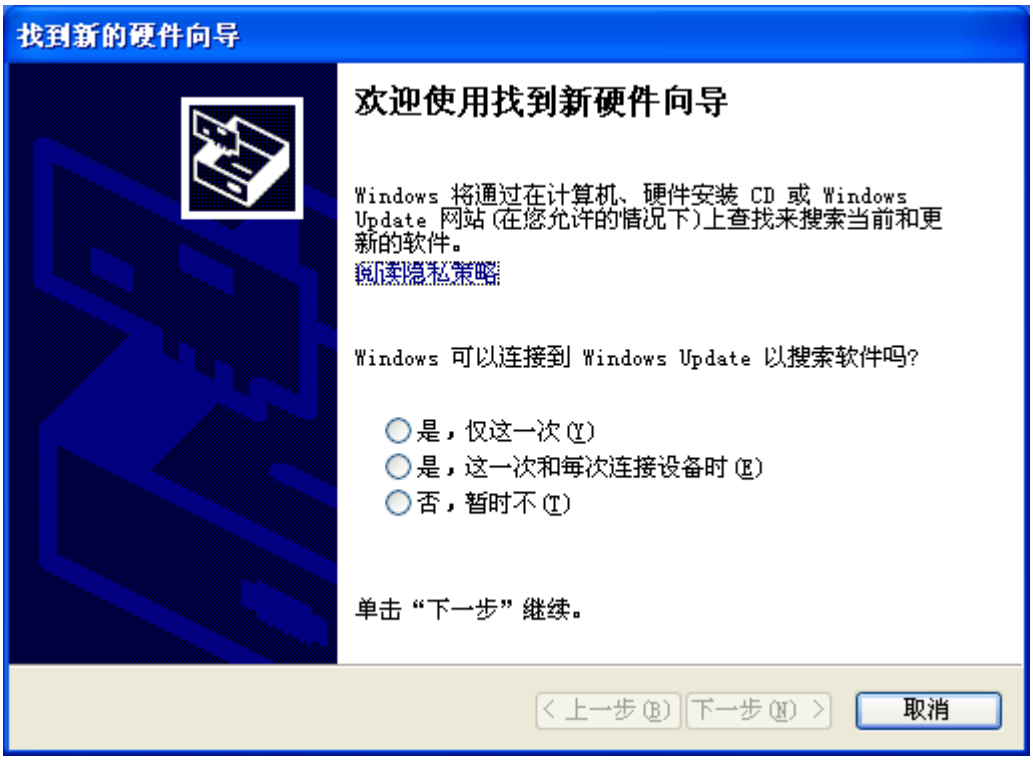
待续

7.10.4 在线安装

待续

7.11 屏幕抓图

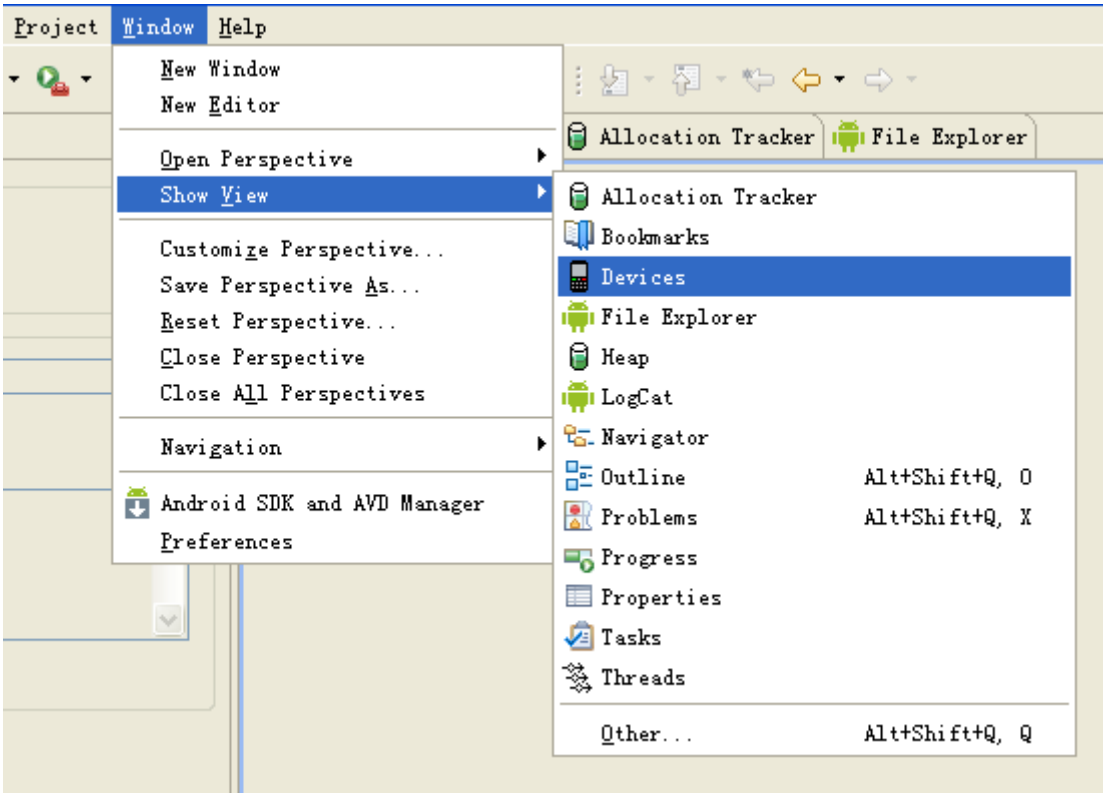
android 有很多截图工具，但是很多都需要 root 权限，在平时做开发时，我们可以使用 eclipse 自带的插件进行截图，非常的方便。使用 USB 延长线将 x210 开发板与 PC 机连接，第一次连接时会提示需要安装软件，如下图：



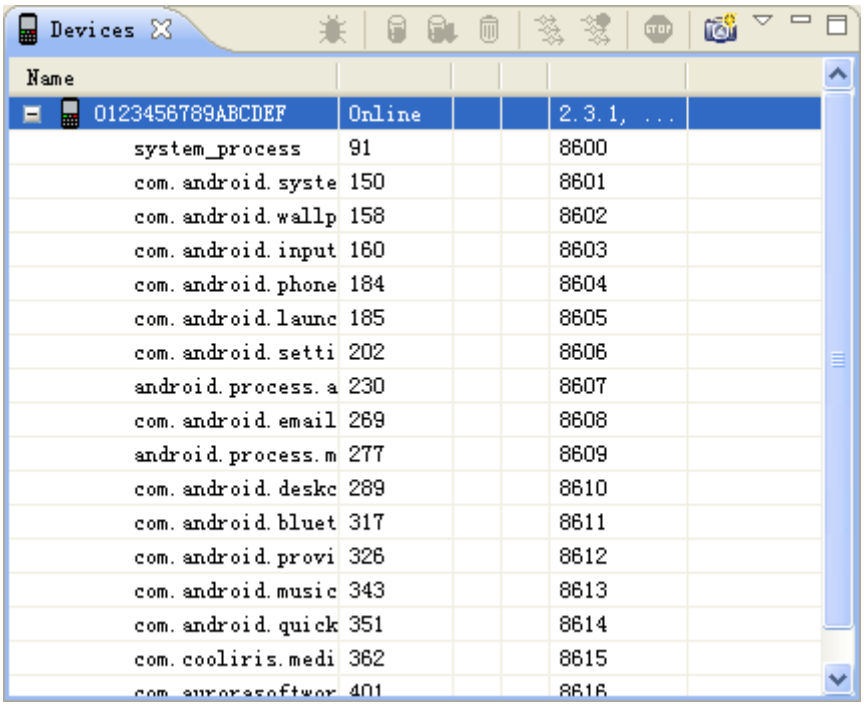
选择否，点下一步：

选择自动安装软件，点下一步，直至安装完成。

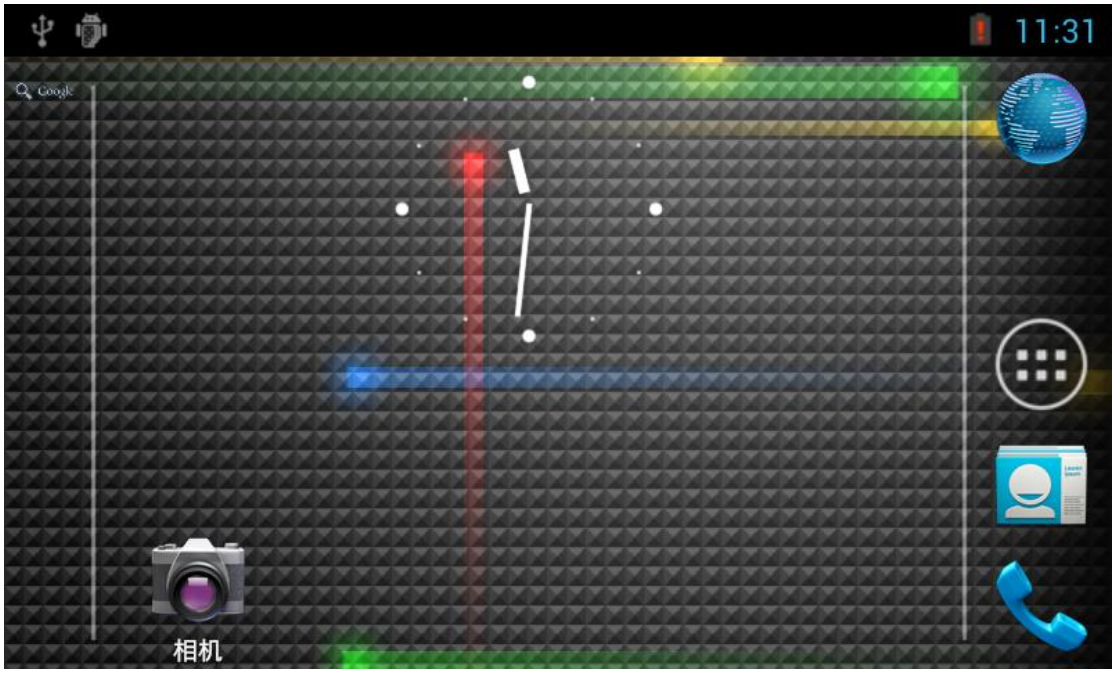
安装完成之后，打开 eclipse 软件，如果没有安装，需先安装该软件。然后点击 Window->Show view->Devices，如下图 ：



确保机器处于开机状态，这时 Devices 会找到机器的设备号，如下图：

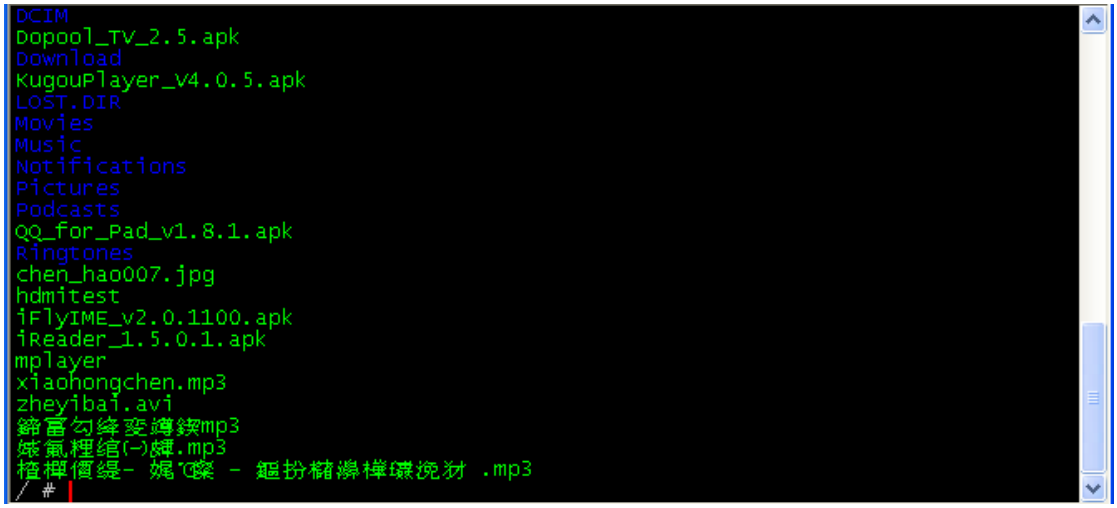


点击上图右上脚的摄相头标志，就会弹出要保存的图像，点击 save 保存即可，如下图所示：



7.12 挂载 SD 卡

系统启动后，会自动挂载右侧卡槽中的 SD 卡到/mnt/sdcard 目录，如下图所示：



7.13 挂载 U 盘

默认插入 U 盘后，系统并没有自动挂载 U 盘，可以通过修改 android 脚本让系统自动挂载 U 盘，也可以手动执行指令挂载，示例指令如下：

```
mount -t vfat /dev/block/sda1 /mnt/usb
```

有些 U 盘没有分区，只有一个设备节点，这时全用如下指令挂载：

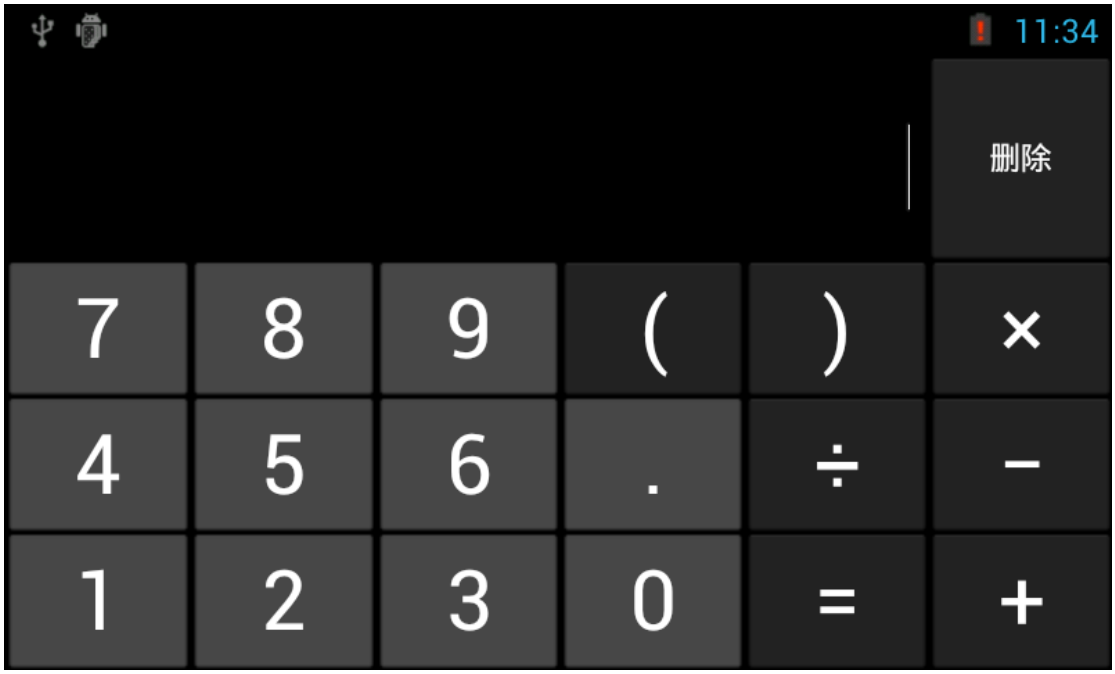
```
mount -t vfat /dev/block/sda1 /mnt/usb
```

成功挂载后界面如下：

```
/ # cd /mnt/
/mnt # ls
asec    ext_sd  obb     sdcard  secure  usb
/mnt # ls usb/
/mnt # cd /
/ # mount -t vfat /dev/block/sda /mnt/usb/
sd 0:0:0:0: [sda] 7856127 512-byte logical blocks: (4.02 GB/3.74 GiB)
sd 0:0:0:0: [sda] No Caching mode page present
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] No Caching mode page present
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda:
/ # ls /mnt/usb/
007.mp4                 autorun.inf             mk
360????                 boot                    pics
??? - ?? - ??????.mp3   casper                  pool
????.mp3                dists                   preseed
?????.mp3               efi                     sensors.x210.so
README.diskdefines      install                 syslinux
SmartClean.ini          ldlinux.sys             uboot_inand.bin
android_ics.tar.bz2     lili.ico                wubi.exe
autorun.bak             md5sum.txt
/ #
```

7.14 计算器

点击 android 应用的计算器即可使用计算器功能，如下图所示：



7.15 命令终端

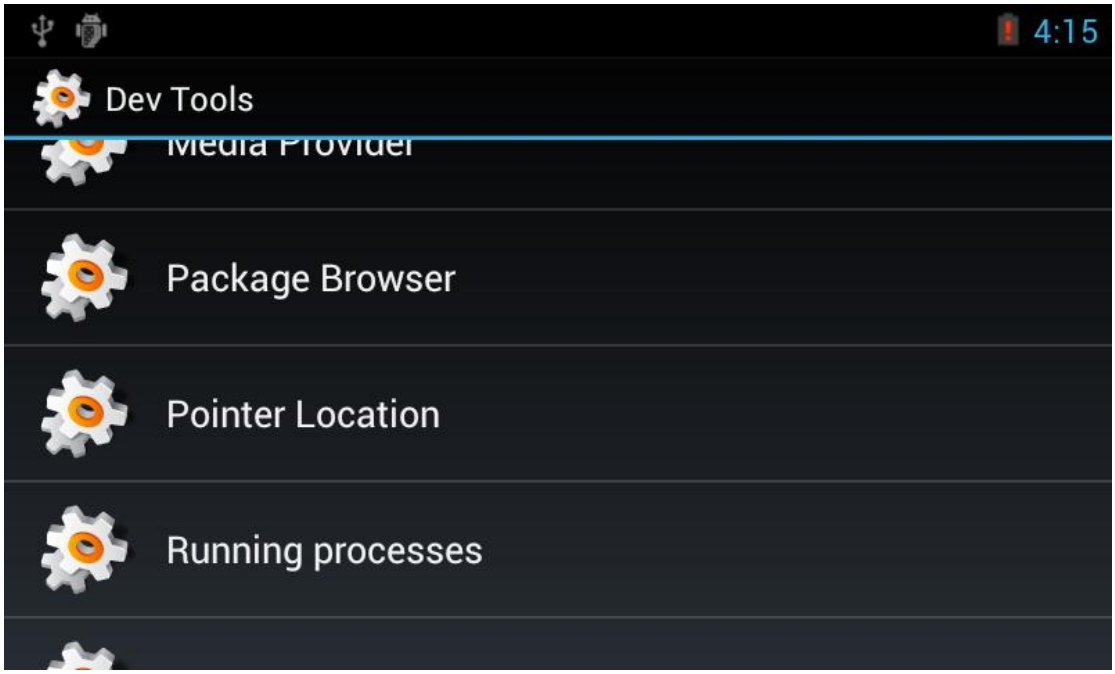
将串口连接电路板，进入 android 系统后，会自动进入 android 终端，如下图所示：

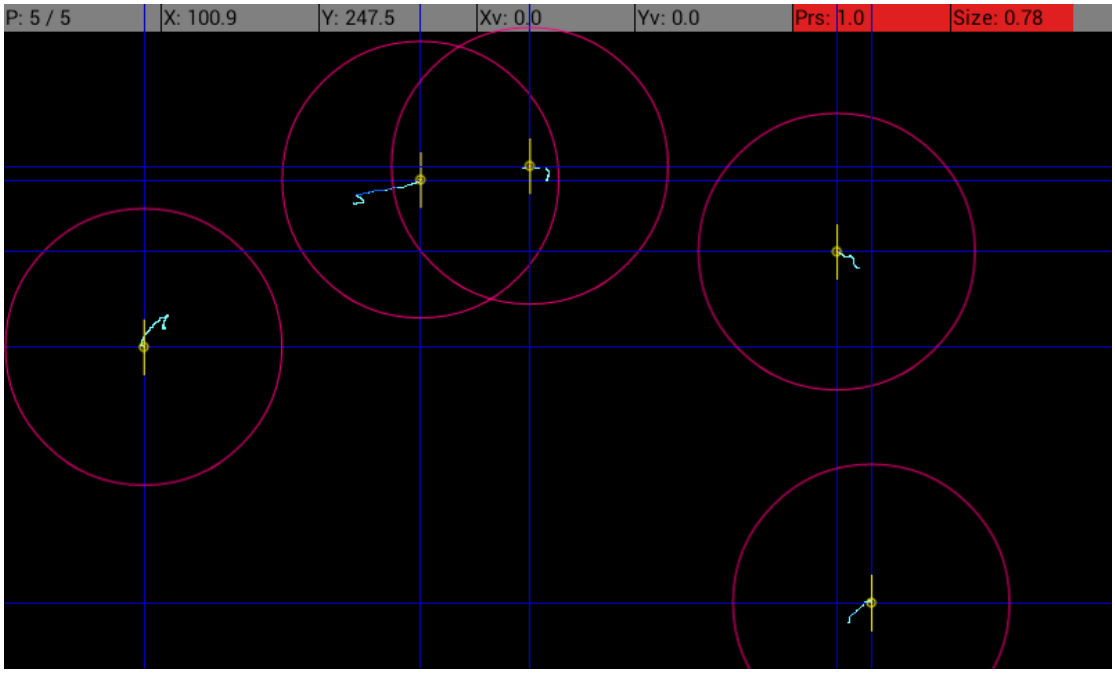
```
/ # ls /mnt/usb/
007.mp4                 autorun.inf             mk
360????                 boot                    pics
??? - ?? - ??????.mp3   casper                  pool
????.mp3                dists                   preseed
?????.mp3               efi                     sensors.x210.so
README.diskdefines      install                 syslinux
SmartClean.ini          ldlinux.sys             uboot_inand.bin
android_ics.tar.bz2     lili.ico                wubi.exe
autorun.bak             md5sum.txt
/ #
/ #
/ #
/ # ls
acct                ext_sd              sbin
cache               init                sdcard
config              init.goldfish.rc    sys
d                   init.rc             system
data                init.x210.rc        ueventd.goldfish.rc
default.prop        init.x210.usb.rc    ueventd.rc
dev                 mnt                 ueventd.x210.rc
etc                 proc                vendor
/ #
```

注意，android 原生系统默认使用的是 toolbox，它相对 busybox 有很多不足之处，我们已经将强大的 busybox 移植到 android4.0 文件系统中。

7.16 电容屏五点触摸

进入 APP 界面，打开 Dev Tools，点击 Pointer Location，再用五指测试：





7.17 输入法

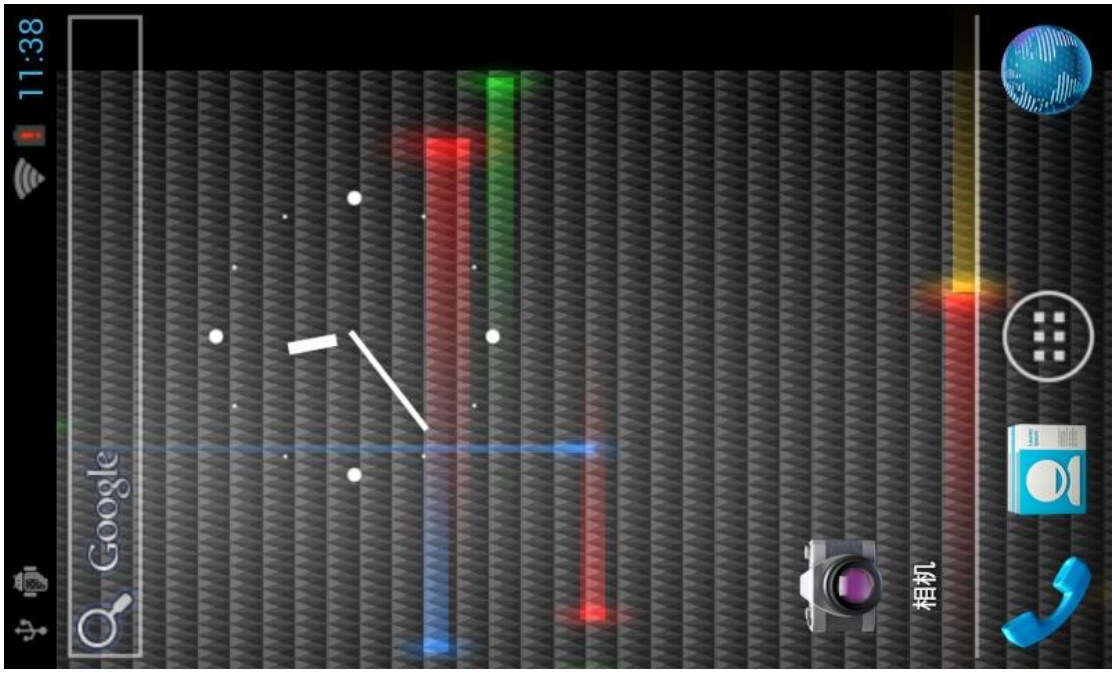
待续

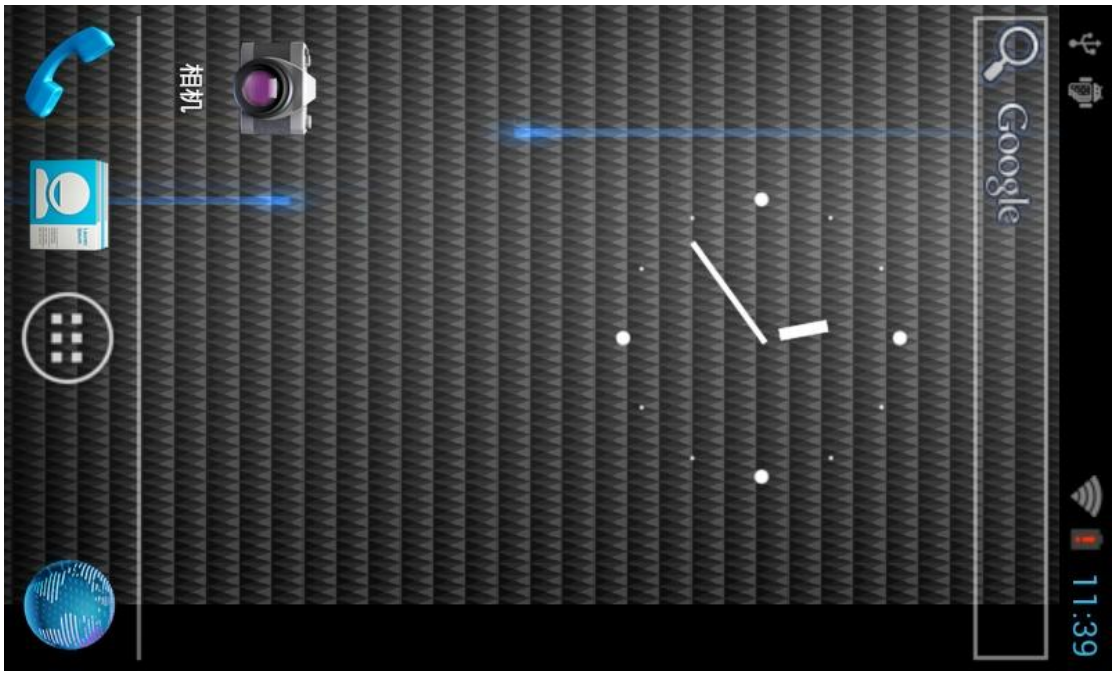
7.18 浏览器

android 默认自带一个浏览器，该浏览器功能已经非常完善了，如果用户仍然觉得不够要求，可以下载安装第三方浏览器，如 UC 浏览器等。

7.19 屏幕旋转

重力传感器已经集成到开发板上，将开发板移动到四周任一方向，界面会随之改变。当然并不是所有应用程序都会随之改变，有部分应用程序不支持屏幕旋转。旋转后示例图片如下：



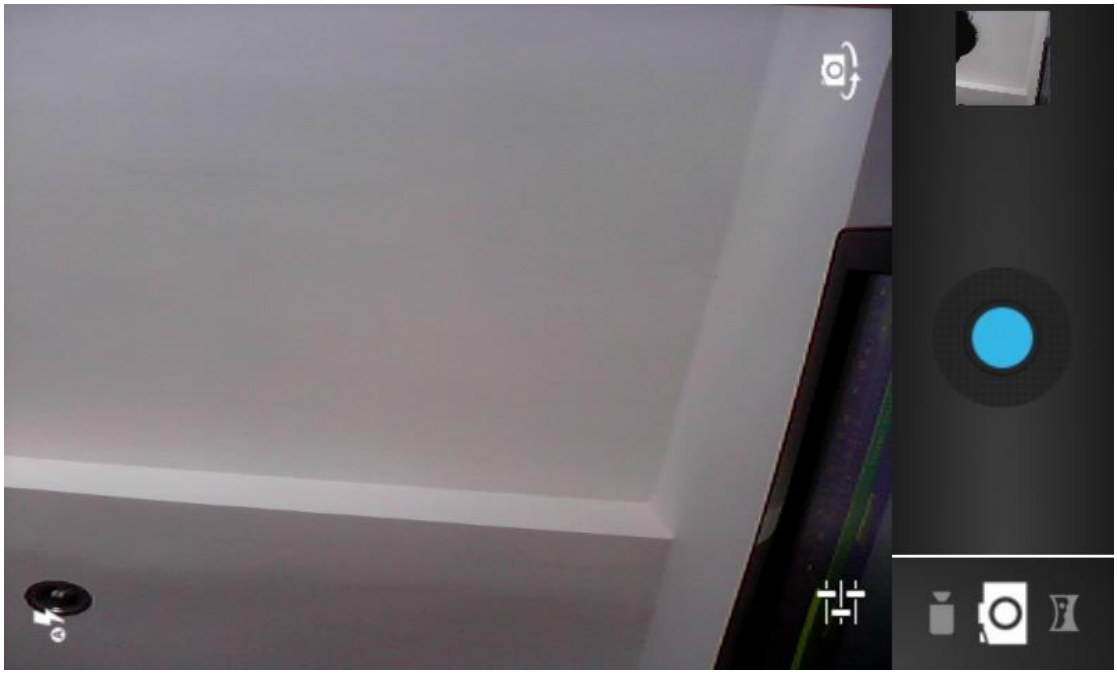


7.20 时间设置

点击 android 应用中的设置，可以看到有日期和时间一栏，点击进去，选择相应的栏目设置即可。

7.21 拍照摄相

点击 android 应用中的相机，会进入图像预览模式。点击右下脚的拍照按钮即可拍照，如下图所示：



右下脚可以切换拍照和录像功能。

7.22 优酷

待续

7.23 播放网页视频

待续

7.24 播放电视

待续

7.25 酷狗

待续

7.26 电子书

待续

7.27 1080P 视频播放

210 支持 MPEG2,MPEG4 等视频的硬解码，可以流畅的播放这些格式的 1080P 视频文件。使用 RockPlayer 或者自带的播放器都可播放。注意，使用 RockPlayer 播放时，一定要选择硬解模式，否则会很卡。播放界面如下：

7.28 QQ

待续

7.29 QQ 斗地主

待续

7.30 愤怒的小鸟

待续

7.31 赛车

用户可自己下载赛车游戏进行测试。

7.32 VGA 显示

注意，x210v3s 已经去掉了 VGA 电路，如需使用 VGA，需另加 VGA 转接板。同时，

将映像文件更新为VGA显示的映像，再使用VGA连接线连接转接板的VGA接口和显示器，或是带VGA接口的电视机，启动开发板即可。目前开发板已测试通过800*600，1024*768，1440*900，1280*1024等分辨率的VGA。

7.33 HDMI显示

HDMI显示支持直接将LCD上显示的视频还原到带有HDMI接口的电视机上，支持1080P高清视频，同时还将音频也一并传送到电视机上。但是有部分电视机HDMI支持的不是很好，可能出现有声音无图像的现象，具体与电视机兼容性有关。

7.34 开关机

待续

7.35 休眠唤醒

x210v3s开发板支持休眠唤醒功能，不仅电容触摸屏支持休眠唤醒，电阻触摸屏同样也支持。在进入系统之后，轻按POWER键，这时整个屏幕全部熄灭，系统进入低功耗模式。再次按下POWER键，将会唤醒机器，进入工作模式。这对于使用电池供电的手持设备来说，是非常有意义的。

注意，如果接上USB OTG后，将无法正常休眠。

第8章 android 内核驱动

8.1 在 android 系统中使用 busybox

默认我们已经将 busybox 移植到 android4.0.4 中。

8.2 G-sensor 驱动

路径：kernel/drivers/char/
文件：x210_sensor_kxtf9.c
说明：该驱动使用查询方式，并没有用到中断管脚。

8.3 电阻触摸屏驱动

路径：kernel/drivers/input/touchscreen
源码：ts-x210.c
说明：校屏软件已经集成到电阻触摸屏驱动中。

8.4 电容触摸屏驱动

路径：kernel/drivers/input/touchscreen
源码：ft5x06_ts.c

8.5 液晶屏驱动

路径：kernel/drivers/video/Samsung，kernel/arch/arm/mach-s5pv210
源码：s3cfb.c，s3cfb.h，mach-x210.c
注意：如需更改液晶屏，主要调节 mach-x210.c 中的相关参数。

8.6 按键驱动

路径：kernel/arch/arm/mach-s5pv210
源码：button-x210.c

8.7 USB 接口 WIFI 驱动

x210v3s 支持 USB WIFI 驱动，购买 USB 接口 WIFI 模块的客户联系客服索取驱动源码及详细移植文档。

8.8 VGA 驱动

关键文件：kernel/arch/arm/mach-s5pv210/mach-x210.c
注意：VGA 或 LCD 输出的选择，通过 menuconfig 里面配置。

8.9 HDMI 驱动

路径：kernel/drivers/media/video/samsung/tv20

8.10 休眠唤醒

8.10.1 android 休眠唤醒过程

android 系统的休眠唤醒，是在标准 linux 休眠唤醒的基础上演变而来。标准 linux 的休眠唤醒过程如下：

标准 linux 休眠过程:

- power management notifiers are executed with PM_SUSPEND_PREPARE
- tasks are frozen
- target system sleep state is announced to the platform-handling code
- devices are suspended
- platform-specific global suspend preparation methods are executed
- non-boot CPUs are taken off-line
- interrupts are disabled on the remaining (main) CPU
- late suspend of devices is carried out (一般有一些 BUS driver 的动作进行)
- platform-specific global methods are invoked to put the system to sleep

标准 linux 唤醒过程:

- the main CPU is switched to the appropriate mode, if necessary
- early resume of devices is carried out (一般有一些 BUS driver 的动作进行)
- interrupts are enabled on the main CPU
- non-boot CPUs are enabled
- platform-specific global resume preparation methods are invoked
- devices are woken up
- tasks are thawed
- power management notifiers are executed with PM_POST_SUSPEND

休眠唤醒相关内核源码：

- kernel/power/main.c
- kernel/power/earlysuspend.c
- kernel/power/wakelock.c
- kernel/power/process.c
- driver/base/power/main.c
- arch/arm/plat-samsung/pm.c

用户可以通过 sys 文件系统查看休眠方式，以及手动让系统休眠等。

```
/ # cd sys
/sys # ls
block       class       devices     fs          module
bus         dev         firmware    kernel      power
/sys # cd power/
/sys/power # ls
pm_async                wait_for_fb_sleep   wake_lock
state                   wait_for_fb_wake    wake_unlock
/sys/power #
```

可以看到，在/sys 目录下，有一个 power 目录，在 power 目录下，有六个文件，我们主要关心 state，它是上层与内核连接的一个桥梁，通过 cat 命令可以查看系统所支持的休眠模式：

```
/sys/power # cat state
mem
/sys/power #
```

可见，所查询的系统仅支持 mem 的挂起方式。通过 echo 命令往 state 中写入数据，手动让系统进入休眠模式：

```
/sys/power # echo mem > state
[ 1636.437795] request_suspend_state: sleep (0->3) at 1635938527752 (2010-01-01 12:27:16.035498876 UTC)
[ 1636.437876] [lqm] early_suspend_work...
/sys/power # [ 1636.649681] PM: Syncing filesystems ... done.
[ 1636.650823] [lqm] ++suspend_prepare
[ 1636.653743] Freezing user space processes ... (elapsed 0.01 seconds) done.
[ 1636.671352] Freezing remaining freezable tasks ... (elapsed 0.01 seconds) done.
[ 1636.686910] [lqm] ++suspend_devices_and_enter,state=3
[ 1636.686956] Suspending console(s) (use no_console_suspend to debug)
```

可以看到，系统已经进入休眠了。

这里是让系统进入休眠的一个入口，android 文件系统让系统进入休眠的实质就是往 state 中写入相关参数实现的。为了弄清楚整个流程，有必要弄清楚上述实现的原理。

在 kernel/power/main.c 中，系统启动时，会启动名为 pm 的工作队列，然后创建/sys/power 目录，紧接着在 power 目录下创建 state 等文件：

```
staticint __init pm_init(void)
{
	int error = pm_start_workqueue();
	if (error)
		return error;
	power_kobj = kobject_create_and_add("power", NULL);//生成/sys/power 目录
	if (!power_kobj)
		return -ENOMEM;
	return sysfs_create_group(power_kobj, &attr_group);//建立 state 等一系列属性文件
}

core_initcall(pm_init);
```

跟踪 attr_group，找到它的定义如下：

```
staticstruct attribute * g[] = {
	&state_attr.attr,
#ifdef CONFIG_PM_TRACE
	&pm_trace_attr.attr,
#endif
#ifdef CONFIG_PM_SLEEP
	&pm_async_attr.attr,
#ifdef CONFIG_PM_DEBUG
	&pm_test_attr.attr,
#endif
#ifdef CONFIG_USER_WAKELOCK
	&wake_lock_attr.attr,
```

```
        &wake_unlock_attr.attr,
#endif
#endif
        NULL,
};

staticstruct attribute_group attr_group = {
        .attrs = g,
};
```

可见，attr_group 是一组文件，具体在 g[]数组中，第一个 state_attr 定义如下：

```
power_attr(state);//申明一个 state_attr 的操作操作,attr 为 state
```

再看看 power_attr 的申明：

```
#define power_attr(_name) \
staticstruct kobj_attribute _name##_attr = {\
        .attr  = {                      \
                .name = __stringify(_name),  \
                .mode = 0644,               \
        },                              \
        .show       = _name##_show,             \
        .store      = _name##_store,        \
}
```

由此，我们可以将 power_attr(state);展开，它相当于定义了如下结构体：

```
#define power_attr(state)
staticstruct kobj_attribute state_attr = {      \
        .attr  = {                      \
                .name = __stringify(state),     \
                .mode = 0644,               \
        },                              \
        .show       = state_show,               \
        .store      = state_store,          \
}
```

即 state 对应有两个可供操作的函数 state_show 和 state_store。state_show 函数用于显示系统支持的睡眠模式：

```
staticssize_tstate_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
        char *s = buf;
#ifdef CONFIG_SUSPEND
        int i;

        for (i = 0; i < PM_SUSPEND_MAX; i++) {
                if (pm_states[i] && valid_state(i))
```

```
                s += sprintf(s,"%s ", pm_states[i]);
    }
#endif
#ifdef CONFIG_HIBERNATION
    s += sprintf(s, "%s\n", "disk");
#else
    if (s != buf)
        /* convert the last space to a newline */
        *(s-1) = '\n';
#endif
    return (s - buf);
}
```

首先通过一个 for 循环，将有效的睡眠状态写到字符指针 s 中，再返回。pm_states 在 suspend.c 中，内容如下：

```
constchar *const pm_states[PM_SUSPEND_MAX] = {
#ifdef CONFIG_EARLYSUSPEND
    [PM_SUSPEND_ON]          = "on",
#endif
    [PM_SUSPEND_STANDBY]     = "standby",
    [PM_SUSPEND_MEM]= "mem",
};
```

这里给出了 on,standby,mem 三种状态，但是前面我们通过 cat 查询，只有 mem 一种，这是因为另外两个被 valid_state 函数过滤掉了。

valid_state 函数原型：

```
boolvalid_state(suspend_state_t state)
{
    /*
     * All states need lowlevel support and need to be valid to the lowlevel
     * implementation, no valid callback implies that none are valid.
     */
    return suspend_ops && suspend_ops->valid&& suspend_ops->valid(state);
}
```

这里我们首先需找到 suspend_ops 结构：

```
staticstruct platform_suspend_ops *suspend_ops;

/**
 *    suspend_set_ops - Set the global suspend method table.
 *    @ops:     Pointer to ops structure.
 */
voidsuspend_set_ops(struct platform_suspend_ops *ops)
{
    mutex_lock(&pm_mutex);
```

```
    suspend_ops = ops;
    mutex_unlock(&pm_mutex);
}
```

可以看到在 suspend_set_ops 函数中，通过传入参数赋给了 suspend_ops。在 arch/arm/plat-samsung/pm.c 中，在 s3c_pm_init 初始化函数中，调用了 suspend_set_ops 函数：

```
int __init s3c_pm_init(void)
{
    printk("S3C Power Management, Copyright 2004 Simtec Electronics\n");
    pmstats_init();

    suspend_set_ops(&s3c_pm_ops);
    return 0;
}
```

在 mach-smdkc110.c 中，调用了 s3c_pm_init 函数。可见，在内核启动的开始，就已经将 s3c_pm_ops 结构赋给了 suspend_ops 结构。s3c_pm_ops 结构如下：

```
staticstruct platform_suspend_ops s3c_pm_ops = {
    .enter          = s3c_pm_enter,
    .prepare   = s3c_pm_prepare,
    .finish         = s3c_pm_finish,
    .valid          = suspend_valid_only_mem,//只允许 mem 挂起
};
```

在 suspend.c 中，定义了 suspend_valid_only_mem 函数：

```
int suspend_valid_only_mem(suspend_state_t state)
{
    return state == PM_SUSPEND_MEM;
}
```

再回到 state_show 函数中，不然理解为什么 cat state 只有 mem 一种睡眠模式了。

回到 main.c 中，在执行 echo mem > state 指令时，在内核中调用了 state_store 函数。

```
staticssize_t state_store(struct kobject *kobj, struct kobj_attribute *attr,
                constchar *buf, size_t n)
{
#ifdef CONFIG_SUSPEND
#ifdef CONFIG_EARLYSUSPEND
    suspend_state_t state = PM_SUSPEND_ON;
#else
    suspend_state_t state = PM_SUSPEND_STANDBY;
#endif
    constchar * const *s;
#endif
    char *p;
    int len;
```

```
    int error = -EINVAL;

    //剥离回车换行符
    p = memchr(buf, '\n', n);
    len = p ? p - buf : n;

    /* First, check if we are requested to hibernate */
    if (len == 4 && !strncmp(buf, "disk", len)) {
        error = hibernate();
goto Exit;
    }

#ifdef CONFIG_SUSPEND
    //查找匹配的状态
    for (s = &pm_states[state]; state < PM_SUSPEND_MAX; s++, state++)
    {
        if (*s && len == strlen(*s) && !strncmp(buf, *s, len))
            break;
    }
    if (state < PM_SUSPEND_MAX && *s)
#ifdef CONFIG_EARLYSUSPEND
        if (state == PM_SUSPEND_ON || valid_state(state))
        {
            error = 0;
            request_suspend_state(state);
        }
#else
        error = enter_state(state);
#endif
#endif

 Exit:
    return error ? error : n;
}
```

buf 是外部写入数据的入口，通过 memchr 函数剥离换行符，然后通过 for 循环查找匹配的状态，如果传入的参数合法，则进入 request_suspend_state 函数请求挂起。可以看出，state 只接受 mem 和 on 两个参数，注意，尽管使用 cat state 只有 mem 一个参数，但是在上面的 if 语句中，或了 on，即唤醒的状态。使用 echo 命令输入其他字符将返回错误。

```
void request_suspend_state(suspend_state_t new_state)
{
    unsignedlong irqflags;
    int old_sleep;
```

```
    spin_lock_irqsave(&state_lock, irqflags);
    old_sleep = state &SUSPEND_REQUESTED;     //state = 1 或者 3
    if (debug_mask &DEBUG_USER_STATE)
    {
        struct timespec ts;
        struct rtc_time tm;
        getnstimeofday(&ts);
        rtc_time_to_tm(ts.tv_sec, &tm);
        pr_info("request_suspend_state: %s (%d->%d) at %lld"
            "(%d-%02d-%02d %02d:%02d:%02d.%09lu UTC)\n",
            new_state != PM_SUSPEND_ON ? "sleep" : "wakeup",
            requested_suspend_state, new_state,
            ktime_to_ns(ktime_get()),
            tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
            tm.tm_hour, tm.tm_min, tm.tm_sec, ts.tv_nsec);
    }
    if (!old_sleep && new_state != PM_SUSPEND_ON)//执行挂起动作
    {
        printk("[lqm] early_suspend_work...\r\n");//lqm
        state |= SUSPEND_REQUESTED;//state =1
        queue_work(suspend_work_queue, &early_suspend_work);
    }
    elseif (old_sleep && new_state == PM_SUSPEND_ON)//执行恢复动作
    {
        printk("[lqm] late_resume_work...\r\n");//lqm
        state &= ~SUSPEND_REQUESTED;    //state =2
        wake_lock(&main_wake_lock);          //对 main_wake_lock 上锁
        queue_work(suspend_work_queue, &late_resume_work);
    }
    requested_suspend_state = new_state;//存储本次休眠或唤醒的状态，供下次休眠唤醒使用
    spin_unlock_irqrestore(&state_lock, irqflags);
}
```

注意， new_state 即前面传过来的参数，这里只会有 on 和 mem 两种可能,分别对应 0 和 3。默认第一次启动时，state=0，old_sleep=0，这时触摸休眠功能时，传入的 new_state＝3,那么!old_sleep && new_state = 1 && 3 = 1 != PM_SUSPEND_ON = 0，故调用 early_suspend_work 的工作队列，执行挂起动作，系统进入休眠。这时，requested_suspend_state 保存当前的工作状态为 3，并赋值给 state，具体代码没有发现在哪里，但是通过打印信息跟踪，state 确实是按照这个逻辑运行的。再次通过外部中断触摸唤醒中断，这时传入的 new_state=0，同时 state 保留上次的工作状态为 3，可以看到 old_sleep=3&1=1，这时!old_sleep&&new_state=0&&0=0，故第一条 if 语句条件不成立，old_sleep&&new_state=1&&0=0＝PM_SUSPEND_ON，这时触发 late_resume_work 的工作队列，系统执行唤醒动作。后面再继续睡眠，唤醒等，反复按上面规则循环。

挂起队列和唤醒队列声明如下：

```
static DECLARE_WORK(early_suspend_work, early_suspend);
static DECLARE_WORK(late_resume_work, late_resume);
```

分别对应 early_suspend 和 late_resume 函数。工作队列 suspend_work_queue 在 wakelock.c 中有如下定义：

```
    suspend_work_queue = create_singlethread_workqueue("suspend");
    if (suspend_work_queue == NULL) {
        ret = -ENOMEM;
        goto err_suspend_work_queue;
    }
```

可以看出，early_suspend_work 和 suspend_work_queue 事实上是被赋予到一个工作队列的线程，使用 queue_work 触发相应的函数。执行挂起程序时，对应代码如下：

```
staticvoid early_suspend(struct work_struct *work)
{
    struct early_suspend *pos;
    unsignedlong irqflags;
    int abort = 0;

    mutex_lock(&early_suspend_lock);
    spin_lock_irqsave(&state_lock, irqflags);
    if (state == SUSPEND_REQUESTED)  //判断是否为请求挂起的状态
        state |= SUSPENDED;
    else  //如果不是请求挂起，则直接退出
        abort = 1;
    spin_unlock_irqrestore(&state_lock, irqflags);

    if (abort)//如果不是请求挂起，则直接退出
    {
        if (debug_mask &DEBUG_SUSPEND)
            pr_info("early_suspend: abort, state %d\n", state);
        mutex_unlock(&early_suspend_lock);
        goto abort;
    }
    //如果是请求挂起状态，则
    //(1)  调用已注册的 early_suspend 函数
    if (debug_mask &DEBUG_SUSPEND)
        pr_info("early_suspend: call handlers\n");
    list_for_each_entry(pos, &early_suspend_handlers, link) {
        if (pos->suspend != NULL)
            pos->suspend(pos);
    }
    //(2)  释放 early_suspend 锁
    mutex_unlock(&early_suspend_lock);
```

```
    if (debug_mask &DEBUG_SUSPEND)
        pr_info("early_suspend: sync\n");
    //(3) 同步文件系统
    sys_sync();
#ifdef CONFIG_CPU_FREQ
    if (has_audio_wake_lock())
        cpufreq_driver_target(cpufreq_cpu_get(0), 800000,
                DISABLE_FURTHER_CPUFREQ);
#endif
abort:
    spin_lock_irqsave(&state_lock, irqflags);
    //(4) 释放 main_wake_lock，进入睡眠模式
    if (state == SUSPEND_REQUESTED_AND_SUSPENDED)
        wake_unlock(&main_wake_lock);
    spin_unlock_irqrestore(&state_lock, irqflags);
}
```

程序再次判断 state 是否为请求挂起状态，这时 state 事实上已经在 **request_suspend_state** 中赋值为请求挂起了，这里估计是为了防止误判。如果确认是请求挂起状态，则调用 list_for_each_entry，调用已注册的 early_suspend 函数，各已经注册 early_suspend 的驱动会依次执行对应的函数，实现驱动外设的挂起。再释放 early_suspend 锁，同步文件系统，最终释放 main_wake_lock 锁，进入睡眠模式。

wake_unlock 函数在 wake_lock.c 中：

```
void wake_unlock(struct wake_lock *lock)
{
    int type;
    unsignedlong irqflags;
    spin_lock_irqsave(&list_lock, irqflags);
    type = lock->flags& WAKE_LOCK_TYPE_MASK;
#ifdef CONFIG_WAKELOCK_STAT
    wake_unlock_stat_locked(lock, 0);
#endif
    if (debug_mask &DEBUG_WAKE_LOCK)
        pr_info("wake_unlock: %s\n", lock->name);
    lock->flags&= ~(WAKE_LOCK_ACTIVE | WAKE_LOCK_AUTO_EXPIRE);
    list_del(&lock->link);    //删除列表中 wake_lock 节点
    list_add(&lock->link, &inactive_locks);
    if (type == WAKE_LOCK_SUSPEND) {
        long has_lock = has_wake_lock_locked(type);
        if (has_lock > 0) //判断 wake_lock 数目是否为 0
        {
            if (debug_mask &DEBUG_EXPIRE)
                pr_info("wake_unlock: %s, start expire timer, "
```

```
                        "%ld\n", lock->name, has_lock);
                mod_timer(&expire_timer, jiffies + has_lock);
            }
            else
            {
                if (del_timer(&expire_timer))
                    if (debug_mask &DEBUG_EXPIRE)
                        pr_info("wake_unlock: %s, stop expire "
                            "timer\n", lock->name);
                if (has_lock == 0)//如果 wake_lock 数目为 0，则执行 suspend_work 的工作队列，
执行 suspend 函数，进入挂起状态
                    queue_work(suspend_work_queue, &suspend_work);
            }
            if (lock == &main_wake_lock)
            {
                if (debug_mask &DEBUG_SUSPEND)
                    print_active_locks(WAKE_LOCK_SUSPEND);
#ifdef CONFIG_WAKELOCK_STAT
                update_sleep_wait_stats_locked(0);
#endif
            }
        }
        spin_unlock_irqrestore(&list_lock, irqflags);
}
EXPORT_SYMBOL(wake_unlock);
```

程序删除列表中的 wake_lock 节点，再判断 wake_lock 的数目是否为 0，如果仍然大于 0，则开启定时器，等待一段时间继续判断，如果为 0，则将 suspend_work 填入挂起的工作队列，代码如下：

```
staticvoid suspend(struct work_struct *work)
{
    int ret;
    int entry_event_num;

    //(1) 先判断当前是否 wake_lock，若有，则退出，若没有，继续往后执行
    if (has_wake_lock(WAKE_LOCK_SUSPEND))
    {
        if (debug_mask &DEBUG_SUSPEND)
            pr_info("suspend: abort suspend\n");
        return;
    }

    entry_event_num = current_event_num;
    //(2) 同步文件系统
```

```
    sys_sync();
    if (debug_mask &DEBUG_SUSPEND)
        pr_info("suspend: enter suspend\n");
    //(3) 调用 pm_suspend 函数
    ret = pm_suspend(requested_suspend_state);
    if (debug_mask &DEBUG_EXIT_SUSPEND)
    {
        struct timespec ts;
        struct rtc_time tm;
        getnstimeofday(&ts);
        rtc_time_to_tm(ts.tv_sec, &tm);
        pr_info("suspend: exit suspend, ret = %d "
            "(%d-%02d-%02d %02d:%02d:%02d.%09lu UTC)\n", ret,
            tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
            tm.tm_hour, tm.tm_min, tm.tm_sec, ts.tv_nsec);
    }
    if (current_event_num == entry_event_num)
    {
        if (debug_mask &DEBUG_SUSPEND)
            pr_info("suspend: pm_suspend returned with no event\n");
        printk("[lqm] wake_lock_timeout: 0.5s\r\n");//lqm.
        wake_lock_timeout(&unknown_wakeup, HZ / 2);
    }
}
static DECLARE_WORK(suspend_work, suspend);
```

程序会再次判断是否存在 wake_lock，如果有，则退出挂起，如果没有，则开始同步文件系统，然后调用 pm_suspend 函数，执行挂起的动作。

```
intpm_suspend(suspend_state_t state)
{
    if (state > PM_SUSPEND_ON && state <= PM_SUSPEND_MAX)
        returnenter_state(state);//进入标准 linux 睡眠过程
    return -EINVAL;
}
EXPORT_SYMBOL(pm_suspend);
```

可见，pm_suspend 函数事实上就是进入了标准的 linux 睡眠过程。

```
intenter_state(suspend_state_t state)
{
    int error;

    if (!valid_state(state))
        return -ENODEV;
```

```
    if (!mutex_trylock(&pm_mutex))
        return -EBUSY;

    printk(KERN_INFO "PM: Syncing filesystems ... ");
    sys_sync();
    printk("done.\n");

    pr_debug("PM: Preparing system for %s sleep\n", pm_states[state]);
    error = suspend_prepare();//冻结进程
    if (error)
        goto Unlock;

    if (suspend_test(TEST_FREEZER))
        goto Finish;

    pr_debug("PM: Entering %s sleep\n", pm_states[state]);
    error = suspend_devices_and_enter(state);//让外设进入休眠，同时唤醒也从这里开始

 Finish:
    pr_debug("PM: Finishing wakeup.\n");
    suspend_finish();//解冻进程和任务，使能 helper 进程，唤醒终端
 Unlock:
    mutex_unlock(&pm_mutex);
    return error;
}
```

程序首先依然是判断是否为有效的睡眠状态，再同步文件系统，接着调用 suspend_prepare 函数冻结进程，最后调用关键的 suspend_devices_and_enter 函数进入休眠。注意，代码就会停在这个函数中，唤醒也是从这里开始的。

```
int suspend_devices_and_enter(suspend_state_t state)
{
    int error;
    gfp_t saved_mask;

    printk("[lqm] ++%s,state=%d¥r¥n",__func__,state);//lqm.

    if (!suspend_ops)
        return -ENOSYS;

    if (suspend_ops->begin)
    {
        error = suspend_ops->begin(state);
        if (error)
            goto Close;
```

```
    }
    suspend_console();// 挂起串口。注：可通过 menuconfig 的 cmd_line 中，添加”no_console_suspend”选项
    saved_mask = clear_gfp_allowed_mask(GFP_IOFS);
    suspend_test_start();
    error = dpm_suspend_start(PMSG_SUSPEND);//调用各驱动的 suspend 函数
    if(error)
    {
        printk(KERN_ERR "PM: Some devices failed to suspend¥n");
        goto Recover_platform;
    }
    suspend_test_finish("suspend devices");
    if (suspend_test(TEST_DEVICES))
        goto Recover_platform;

    suspend_enter(state);//进入睡眠，同时从这里唤醒

 Resume_devices:
    suspend_test_start();
    dpm_resume_end(PMSG_RESUME);
    suspend_test_finish("resume devices");
    set_gfp_allowed_mask(saved_mask);
    resume_console();//恢复控制栏
 Close:
    if (suspend_ops->end)
        suspend_ops->end();
    return error;

 Recover_platform:
    if (suspend_ops->recover)
        suspend_ops->recover();
    goto Resume_devices;
}
```

挂起程序走到这里，会调用 suspend_console();至此，串口终端将停止显示打印信息。然后调用 dpm_suspend_start 函数挂起各驱动外设，再在 suspend_enter 函数中进入睡眠。

```
staticintsuspend_enter(suspend_state_t state)
{
    int error;

    if (suspend_ops->prepare)
    {
        error = suspend_ops->prepare();
        if (error)
```

```
            return error;
    }

    error = dpm_suspend_noirq(PMSG_SUSPEND);
    if (error) {
        printk(KERN_ERR "PM: Some devices failed to power down\n");
        goto Platfrom_finish;
    }

    if (suspend_ops->prepare_late) {
        error = suspend_ops->prepare_late();
        if (error)
            goto Power_up_devices;
    }

    if (suspend_test(TEST_PLATFORM))
        goto Platform_wake;

    error = disable_nonboot_cpus();//关闭多 CPU 中的非启动 CPU
    if (error || suspend_test(TEST_CPUS))
        goto Enable_cpus;

    arch_suspend_disable_irqs();//关闭 IRQ
    BUG_ON(!irqs_disabled());

    error = sysdev_suspend(PMSG_SUSPEND);//挂起所有的系统设备
    if (!error)
    {
        if (!suspend_test(TEST_CORE))
            error = suspend_ops->enter(state);//CPU 进入省电状态，代码的执行停在这里
        sysdev_resume();//唤醒 CPU
    }

    arch_suspend_enable_irqs();//使能 IRQ
    BUG_ON(irqs_disabled());

Enable_cpus:
    enable_nonboot_cpus();//使能非启动 CPU

Platform_wake:
    if (suspend_ops->wake)
        suspend_ops->wake();
```

```
 Power_up_devices:
     dpm_resume_noirq(PMSG_RESUME);

 Platfrom_finish:
     if (suspend_ops->finish)
         suspend_ops->finish();

     return error;
}
```

这里程序先关闭中断，关闭多 CPU 中的非启动 CPU，挂起所有的系统设备，再执行 suspend_ops->enter(state)，CPU 将进入省电状态，代码也就会留在这段函数里面。suspend_ops 结构体前面已经提到被赋值给 s3c_pm_ops，可见 suspend_ops->enter(state) 对应的函数为 s3c_pm_enter。

```
staticints3c_pm_enter(suspend_state_t state)
{
     staticunsignedlong regs_save[16];

     /* ensure the debug is initialised (if enabled) */

     s3c_pm_debug_init();

     S3C_PMDBG("%s(%d)\n", __func__, state);
     printk("[lqm] %s(%d)\n", __func__, state);//lqm added.

     if (pm_cpu_prep == NULL || pm_cpu_sleep == NULL)
     {
         printk(KERN_ERR "%s: error: no cpu sleep function\n", __func__);
         return -EINVAL;
     }

     /* check if we have anything to wake-up with... bad things seem
      * to happen if you suspend with no wakeup (system will often
      * require a full power-cycle)
     */

     if (!any_allowed(s3c_irqwake_intmask, s3c_irqwake_intallow) &&
         !any_allowed(s3c_irqwake_eintmask, s3c_irqwake_eintallow))
     {
         printk(KERN_ERR "%s: No wake-up sources!\n", __func__);
         printk(KERN_ERR "%s: Aborting sleep\n", __func__);
         return -EINVAL;
     }
```

```
/* store the physical address of the register recovery block */
// resg_save 没有赋值？
s3c_sleep_save_phys = virt_to_phys(regs_save);

S3C_PMDBG("s3c_sleep_save_phys=0x%08lx\n", s3c_sleep_save_phys);
printk("[lqm] s3c_sleep_save_phys=0x%08lx\n", s3c_sleep_save_phys);//lqm added.

/* save all necessary core registers not covered by the drivers */
s3c_pm_save_gpios();
s3c_pm_save_uarts();
s3c_pm_save_core();

s5pv210_platform_enter_io_sleep();

/* set the irq configuration for wake */
s3c_pm_configure_extint();

S3C_PMDBG("sleep: irqwakeup masks: %08lx,%08lx\n",
      s3c_irqwake_intmask, s3c_irqwake_eintmask);
printk("[lqm] sleep: irqwakeup masks: %08lx,%08lx\n",
            s3c_irqwake_intmask, s3c_irqwake_eintmask);//lqm added

s3c_pm_arch_prepare_irqs();

/* call cpu specific preparation */
pm_cpu_prep();

/* flush cache back to ram */
flush_cache_all();
s3c_pm_check_store();

/* clear wakeup_stat register for next wakeup reason */
__raw_writel(__raw_readl(S5P_WAKEUP_STAT), S5P_WAKEUP_STAT);

/* send the cpu to sleep... */
s3c_pm_arch_stop_clocks();

/* s3c_cpu_save will also act as our return point from when
 * we resume as it saves its own register state and restores it
 * during the resume.    */

pmstats->sleep_count++;
pmstats->sleep_freq = __raw_readl(S5P_CLK_DIV0);
```

```
	s3c_cpu_save(regs_save);
	pmstats->wake_count++;
	pmstats->wake_freq = __raw_readl(S5P_CLK_DIV0);

	/* restore the cpu state using the kernel's cpuinit code. */
	cpu_init();

	fiq_glue_resume();
	local_fiq_enable();

	s3c_pm_restore_core();
	s3c_pm_restore_uarts();
	s3c_pm_restore_gpios();
	s5pv210_restore_eint_group();

	s3c_pm_debug_init();

/* restore the system state */

	if (pm_cpu_restore)
		pm_cpu_restore();

	/* check what irq (if any) restored the system */

	s3c_pm_arch_show_resume_irqs();

	S3C_PMDBG("%s: post sleep, preparing to return\n", __func__);
	printk("[lqm] %s: post sleep, preparing to return\n", __func__);//lqm added

	/* LEDs should now be 1110 */
	s3c_pm_debug_smdkled(1 << 1, 0);
	s3c_pm_check_restore();
	s3c_pm_restore_init();

	/* ok, let's return from sleep */
	S3C_PMDBG("S3C PM Resume (post-restore)\n");
	printk("[lqm] S3C PM Resume (post-restore)\n");//lqm added
	return 0;
}
```

程序第一句初始化 pm 的调试，即允许在这里打印信息，只需将相应的宏打开即可，这里我们不用理会。紧接着检查唤醒源，存储恢复寄存器的物理地址，保存 GPIO，串口的现场，再 IO 口设置为休眠模式，调用 s3c_pm_configure_extint 函数配置唤醒源的外部中断口：

```
void s3c_pm_configure_extint(void)
{
	s3c_gpio_setpull(S5PV210_GPH0(1), S3C_GPIO_PULL_UP);
	s3c_gpio_cfgpin(S5PV210_GPH0(1), S3C_GPIO_SFN(0xf));

	enable_irq_wake(IRQ_EINT(1));
	//enable_irq(IRQ_EINT(1));
}
```

这里唤醒中断源为 EINT1，故需将 EINT1 管脚设置为外部中断，且使能中断。

接着设置中断相关寄存器，调用 pm_cpu_prep()函数，在 mach-s5pv210/pm.c 中，有如下定义：

```
static __init int s5pv210_pm_drvinit(void)
{
	pm_cpu_prep = s5pv210_pm_prepare;
	pm_cpu_sleep = s5pv210_cpu_suspend;
	pm_cpu_restore = s5pv210_pm_resume;
	return 0;
}
```

可以跟到对应的函数 s5pv210_pm_prepare 中，实质上是对休眠唤醒相关寄存器进行设置。在执行 s3c_pm_arch_stop_clocks 函数后，CPU 进入睡眠。

在外界中断触发唤醒后，从 cpu_init()函数开始唤醒。唤醒后，会恢复中断，恢复串口，恢复 GPIO 等。

再回到 suspend.c 的 **suspend_enter** 函数中，前面提到系统会停留在 error = suspend_ops->enter(state);语句中，在唤醒后，程序依然会从这里往后执行，紧接着调用 sysdev_resume 函数唤醒 CPU,接着使能 IRQ,使能非启动 CPU,再调用 suspend_ops->finish()函数结束挂起事件，suspend_ops->finish()函数对应于 **s3c_pm_finish 函数**。然后程序进入唤醒剩下的流程。

再回到 **request_suspend_state** 函数，在系统收到唤醒中断事件后，会给出一个唤醒的请求，这时该函数的传入参数 new_state 为 0，像上次一样，将 new_state=0 代进函数方程式，得出最终程序会触发 late_resume_work 的队列，即 **late_resume** 函数得到执行。其代码如下：

```
staticvoid late_resume(struct work_struct *work)
{
	struct early_suspend *pos;
	unsignedlong irqflags;
	int abort = 0;

	mutex_lock(&early_suspend_lock);
	spin_lock_irqsave(&state_lock, irqflags);
	if (state == SUSPENDED)
		state &= ~SUSPENDED;
	else
```

```
            abort = 1;
        spin_unlock_irqrestore(&state_lock, irqflags);

        if (abort)
        {
            if (debug_mask &DEBUG_SUSPEND)
                pr_info("late_resume: abort, state %d\n", state);
            goto abort;
        }
#ifdef CONFIG_CPU_FREQ
        if (has_audio_wake_lock())
            cpufreq_driver_target(cpufreq_cpu_get(0), 800000,
                    ENABLE_FURTHER_CPUFREQ);

#endif
        if (debug_mask &DEBUG_SUSPEND)
            pr_info("late_resume: call handlers\n");
        list_for_each_entry_reverse(pos, &early_suspend_handlers, link)
            if (pos->resume != NULL)
                pos->resume(pos);
        if (debug_mask &DEBUG_SUSPEND)
            pr_info("late_resume: done\n");

abort:
        mutex_unlock(&early_suspend_lock);
}
```

程序开始即清除 state 原来的挂起状态，再调用各驱动中相应的挂起后的恢复函数，完成唤醒功能。

8.10.2 android 系统的 wake_lock 机制

wakelock 有 3 种类型，常用为 WAKE_LOCK_SUSPEND，作用是防止系统进入睡眠。其他类型不是很清楚。定义玫举在 kernel/include/linux/wakelock.h 中：

```
enum {
        WAKE_LOCK_SUSPEND, /* Prevent suspend */
        WAKE_LOCK_IDLE,         /* Prevent low power idle */
        WAKE_LOCK_TYPE_COUNT
};
```

Wakelock 有加锁和解锁 2 种操作，加锁有 2 种方式，第一种是永久加锁（wake_lock），这种锁必须手动的解锁；另一种是超时锁（wake_lock_timeout），这种锁在过去指定时间后，会自动解锁。永久加锁代码如下：

```
voidwake_lock(struct wake_lock *lock)
{
        wake_lock_internal(lock, 0, 0);
```

```
}
EXPORT_SYMBOL(wake_lock);
```

超时锁代码如下：

```
void wake_lock_timeout(struct wake_lock *lock, long timeout)
{
    wake_lock_internal(lock, timeout, 1);
}
EXPORT_SYMBOL(wake_lock_timeout);
```

可见，二者仅传入参数不一样。对于 wakelock，timeout = has_timeout = 0；直接加锁后，然后退出。

```
staticvoid wake_lock_internal(
    struct wake_lock *lock, long timeout, int has_timeout)
{
    int type;
    unsignedlong irqflags;
    long expire_in;

    spin_lock_irqsave(&list_lock, irqflags);
    type = lock->flags & WAKE_LOCK_TYPE_MASK;
    BUG_ON(type >= WAKE_LOCK_TYPE_COUNT);
    BUG_ON(!(lock->flags & WAKE_LOCK_INITIALIZED));
#ifdef CONFIG_WAKELOCK_STAT
    if (type == WAKE_LOCK_SUSPEND && wait_for_wakeup) {
        if (debug_mask & DEBUG_WAKEUP)
            pr_info("wakeup wake lock: %s\n", lock->name);
        wait_for_wakeup = 0;
        lock->stat.wakeup_count++;
    }
    if ((lock->flags & WAKE_LOCK_AUTO_EXPIRE) &&
        (long)(lock->expires - jiffies) <= 0) {
        wake_unlock_stat_locked(lock, 0);
        lock->stat.last_time = ktime_get();
    }
#endif
    if (!(lock->flags & WAKE_LOCK_ACTIVE)) {
        lock->flags |= WAKE_LOCK_ACTIVE;
#ifdef CONFIG_WAKELOCK_STAT
        lock->stat.last_time = ktime_get();
#endif
    }
    list_del(&lock->link);
    if (has_timeout) {
```

```
            if (debug_mask &DEBUG_WAKE_LOCK)
                pr_info("wake_lock: %s, type %d, timeout %ld.%03lu\n",
                    lock->name, type, timeout / HZ,
                    (timeout % HZ) * MSEC_PER_SEC / HZ);
            lock->expires = jiffies + timeout;
            lock->flags |= WAKE_LOCK_AUTO_EXPIRE;
            list_add_tail(&lock->link, &active_wake_locks[type]);
        } else {
            if (debug_mask &DEBUG_WAKE_LOCK)
                pr_info("wake_lock: %s, type %d\n", lock->name, type);
            lock->expires = LONG_MAX;
            lock->flags &= ~WAKE_LOCK_AUTO_EXPIRE;
            list_add(&lock->link, &active_wake_locks[type]);
        }
        if (type == WAKE_LOCK_SUSPEND) {
            current_event_num++;
#ifdef CONFIG_WAKELOCK_STAT
            if (lock == &main_wake_lock)
                update_sleep_wait_stats_locked(1);
            elseif (!wake_lock_active(&main_wake_lock))
                update_sleep_wait_stats_locked(0);
#endif
            if (has_timeout)
                expire_in = has_wake_lock_locked(type);
            else
                expire_in = -1;
            if (expire_in > 0) {
                if (debug_mask &DEBUG_EXPIRE)
                    pr_info("wake_lock: %s, start expire timer, "
                        "%ld\n", lock->name, expire_in);
                mod_timer(&expire_timer, jiffies + expire_in);
            } else {
                if (del_timer(&expire_timer))
                    if (debug_mask &DEBUG_EXPIRE)
                        pr_info("wake_lock: %s, stop expire timer\n",
                            lock->name);
                if (expire_in == 0)
                    queue_work(suspend_work_queue, &suspend_work);
            }
        }
        spin_unlock_irqrestore(&list_lock, irqflags);
}
```

而对于wake_lock_timeout，在经过timeout时间后，才加锁。再判断当前持有wakelock时，

启动另一个定时器，在expire_timer的回调函数中再次判断是否持有wakelock。

```
staticvoid expire_wake_locks(unsignedlong data)
{
	long has_lock;
	unsignedlong irqflags;
	if (debug_mask &DEBUG_EXPIRE)
		pr_info("expire_wake_locks: start\n");
	spin_lock_irqsave(&list_lock, irqflags);
	if (debug_mask &DEBUG_SUSPEND)
		print_active_locks(WAKE_LOCK_SUSPEND);
	has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
	if (debug_mask &DEBUG_EXPIRE)
		pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
	if (has_lock == 0)
		queue_work(suspend_work_queue, &suspend_work);
	spin_unlock_irqrestore(&list_lock, irqflags);
}
static DEFINE_TIMER(expire_timer, expire_wake_locks, 0, 0);
```

在 wakelock 中，有 2 个地方可以让系统从 early_suspend 进入 suspend 状态。分别是：
a: 在 wake_unlock 中，解锁之后，若没有其他的 wakelock，则进入 suspend。
b: 在超时锁的定时器超时后，定时器的回调函数，会判断有没有其他的wakelock，若没有，则进入suspend。

8.11 proc 文件系统

8.11.1 启动环境变量查询

使用如下指令查询启动环境变量配置：

```
cat /proc/cmdline
```

会有如下类似打印信息：

```
console=ttySAC2,115200 init=/init
```

8.11.2 CPU 信息查询

使用如下指令查询 CPU 信息：

```
cat /proc/cpuinfo
```

会有如下类似打印信息：

```
/ # cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 2 (v7l)
BogoMIPS        : 992.87
Features        : swp half thumb fastmult vfp edsp neon vfpv3
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x2
CPU part        : 0xc08
```

```
CPU revision    : 2

Hardware        : X210
Revision        : 0000
Serial          : 0000000000000000
```

8.11.3 内存信息查询

使用如下指令查询内存使用信息：

```
cat  /proc/meminfo
```

会有如下类似打印信息：

```
/ # cat /proc/meminfo
MemTotal:         390468 kB
MemFree:           57668 kB
Buffers:             916 kB
Cached:           137580 kB
SwapCached:            0 kB
Active:           168496 kB
Inactive:          82472 kB
Active(anon):     112496 kB
Inactive(anon):      272 kB
Active(file):      56000 kB
Inactive(file):    82200 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:        112500 kB
Mapped:           104152 kB
Shmem:               300 kB
Slab:              9720 kB
SReclaimable:      3628 kB
SUnreclaim:        6092 kB
KernelStack:       3008 kB
PageTables:        5612 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      195232 kB
Committed_AS:    4029244 kB
VmallocTotal:     204800 kB
VmallocUsed:       49796 kB
```

```
VmallocChunk:        131708 kB
```

8.11.4 磁盘分区信息查询

使用如下命令查询磁盘分区信息：

```
cat   /proc/partitions
```

会有如下类似打印信息：

```
/ # cat /proc/partitions
major minor  #blocks  name

 179        0    3891200 mmcblk0
 179        1    3140865 mmcblk0p1
 179        2     262372 mmcblk0p2
 179        3     361237 mmcblk0p3
 179        4     102667 mmcblk0p4
 179       16       2048 mmcblk0boot1
 179        8       2048 mmcblk0boot0
 179       24    3872256 mmcblk1
 179       25    3871232 mmcblk1p1
```

8.11.5 内核版本查询

使用如下命令查询内核版本：

```
cat   /proc/version
```

会有如下类似打印信息：

```
/ # cat /proc/version
Linux version 3.0.8 (jjj@ubuntu-server) (gcc version 4.4.1 (Sourcery G++ Lite 2009q3-67) ) #126
PREEMPT Fri Nov 23 14:16:12 CST 2012
```

8.11.6 网络设备查询

使用如下命令查询网络设备信息：

```
cat   /proc/net/dev
```

会有如下类似打印信息：

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
    lo:    0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
  eth0:    0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
  usb0:    0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
  sit0:    0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
```

8.11.7 查看内核启动信息

使用如下命令查询内核启动信息：

```
cat  /proc/kmsg
```

第9章 x210v3s 项目实战

9.1 x210v3s nand flash 和 inand 启动速度对照表

x210v3 开发平台 nand flash 与 inand 各操作系统启动速度对照表				
	android2.3	android4.0	QT4.8	QTOPIA
nand	50 秒	74 秒	70 秒	30 秒
inand	18 秒	23 秒	11 秒	7 秒

9.2 x210v3s 各操作系统启动参数说明

x210v3 开发平台 nand flash 与 inand 各操作系统启动参数设置		
nand	android2.3	setenv bootargs "root=/dev/mtdblock4 rootfstype=yaffs2 init=/init console=ttySAC0,115200" setenv bootcmd "nand read C0008000 600000 400000;bootm C0008000"
	android4.0	setenv bootcmd "nand read C0008000 600000 400000;bootm C0008000"
	QT4.8	setenv bootargs "root=/dev/mtdblock4 rw init=/linuxrc rootfstype=jffs2 console=ttySAC2,115200" setenv bootcmd "nand read C0008000 600000 500000; bootm C0008000"
	QTOPIA	同 QT4.8
inand	android2.3	setenv bootcmd "movi read kernel 30008000;bootm 30008000"
	android4.0	setenv bootcmd "movi read kernel 30008000;bootm 30008000"
	QT4.8	setenv bootcmd "movi read kernel 30008000;bootm 30008000" setenv bootargs "console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3"
	QTOPIA	同 QT4.8

9.3 x210v3s inand 平台 android2.3,android4.0,qt 系统如何互刷操作系统

由于这几个操作系统用的 uboot 有些差异，分区也不同，因此互刷系统时，要重新制作分区表。QT4.8.3 和 QTOPIA 系统不需要重刷，二者 uboot 和内核完全相同。

注意，android4.0 和 QT 使用的调试串口 2，android2.3 和 WINCE 使用的调试串口 0.

第一步：使用当前的 uboot 启动，进入控制台后执行 fastboot

第二步：在 ubuntu 系统下开启另一个命令行终端，或在 xp 下打开 dos 界面执行窗口，执行如下指令更新最新的 uboot:

```
fastboot  flash  bootloader  uboot.bin
```

第三步：重启开发板，运行最新的 uboot，执行如下指令重新分区：

```
fdisk  -c  0
```

第四步：使用如下指令更新剩下的映像：

android 平台：

```
fastboot  flash  kernel  zImage
fastboot  flash  system  x210.img
```

QT 平台：

```
fastboot  flash  kernel  zImage
fastboot  flash  system  rootfs_qt*.ext3[QT4.8 和 QTOPIA 名称有差异，指定名称]
```

第10章 其他产品介绍

10.1 核心板系列

X6410CV10
X210CV3
X210CV4
G210CV10
I210CV20
X4412CV2
X4418CV2
X6818CV3
X3288CV3

10.2 开发板系列

x6410 开发板
x210 开发板
g210 开发板
i210 开发板
x4412 开发板
x4418 开发板
X6818 开发板
X3288 开发板
ibox4412 卡片电脑
ibox4418 卡片电脑
ibox6818 卡片电脑

说明：产品详细规格，以及更多其他产品请关注九鼎创展官方网站和论坛。