Linux Kernel and Android Development Class

Lab Book

Free Electrons, Adeneo embedded
http://free-electrons.com
http://adeneo-embedded.com

December 6, 2012

About this document

This document can be found on http://www.adeneo-embedded.com/.

It was generated from LaTeX sources found on http://git.free-electrons.com/training-materials.

More details about our training sessions can be found on http://free-electrons.com/training.

Copying this document

© 2004-2012, Free Electrons, http://free-electrons.com, Adeneo Embedded, http://adeneo-embedded.com.

This document is released under the terms of the Creative Commons CC BY-SA 3.0 license . This means that you are free to download, distribute and even modify it, under certain conditions.

Corrections, suggestions, contributions and translations are welcome!

Lab 1 : Compiling/Running a Linux kernel

Objective: Learn how to flash and program an environment for the target board. Learn how to cross-compile a Linux kernel to target the PandaBoard

After this labs you will be able to :

- Flash full system on the development board
- Launch the embedded Linux
- Play with the demonstration image
- Set up a cross-compiling environment
- Configure the kernel Makefile accordingly
- Cross compile the kernel for the PandaBoard platform
- Check that the kernel you compiled can boot onto the target

For first contact with the PandaBoard, we choose to boot the Linux image using a SDCard. A SDCard needs to be prepared before the Linux image can be copied and booted by the board.

WARNING:All data previously stored on the SDCard will be destroyed. Make sure you backup its content before continuing this lab.

Format the SDcard

The steps below explain how to format the SDCard. Plug the SDCard into your computers card reader and find out the device name of the SD with the following command:

```
$ dmesg
...
[ 840.957799] sdb: sdb1
[ 841.075159] sd 3:0:0:0: [sdb] Assuming drive cache: write through
[ 841.075167] sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

In the example above, the SDCard device name is `/dev/sdb`. We will use this name throughout this lab. Make sure you replace this name by the one suitable for your computer every time `/dev/sdb` is mentioned in this lab. Check if your distribution automatically mounted the SDCard, and unmount it if it is the case:

```
$ mount
.....
/dev/sdb1 on /media/D21A-8E43 type vfat
(rw,nosuid,nodev,uhelper=udisks,uid=1000,gid=1000,shortname=mixe
d,dmask=0077,utf8=1,flush)
$ sudo umount /dev/sdb1
```

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Launch `fdisk` to create the partitions.

```
$ sudo fdisk /dev/sdb
WARNING: DOS-compatible mode is deprecated. It's strongly
recommended to
switch off the mode (command 'c') and change display units to
sectors (command 'u').
Command (m for help): m
Command action
a toggle a bootable flag
b edit bsd disklabel
c toggle the dos compatibility flag
d delete a partition
l list known partition types
m print this menu
n add a new partition
o create a new empty DOS partition table
p print the partition table
q quit without saving changes
s create a new empty Sun disklabel
t change a partition's system id
u change display/entry units
v verify the partition table
w write table to disk and exit
x extra functionality (experts only)
Command (m for help): p
Disk /dev/sdb: 252 MB, 252968960 bytes
255 heads, 63 sectors/track, 30 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0067f7b9
Device Boot
/dev/sdb1
Start
1
End
31
Blocks Id System
247008+ c W95 FAT32 (LBA)
Partition 1 has different physical/logical endings:
phys=(29, 254, 63) logical=(30, 192, 34)
```

Write down the total size value. Enter the expert mode to set the geometry:

```
Command (m for help): x
Expert command (m for help): h
Number of heads (1-256, default 255): 255
Expert command (m for help): s
Number of sectors (1-63, default 63): 63
Warning: setting sector offset for DOS compatiblity
```

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Compute the number of cylinders by dividing the total size noted earlier by `(255x63x512)`. In this example, it would give: `252968960/255/63/512=30,75` rounded down to `30`. Enter the number of cylinders:

```
Expert command (m for help): c
Number of cylinders (1-1048576, default 30): 30
Expert command (m for help): r
Command (m for help): p
Disk /dev/sdb: 252 MB, 252968960 bytes
255 heads, 63 sectors/track, 30 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0067f7b9
Device Boot
/dev/sdb1
Start
1
End
31
Blocks Id System
247008+ c W95 FAT32 (LBA)
Partition 1 has different physical/logical endings:
phys=(29, 254, 63) logical=(30, 192, 34)
```

If a partition already exists, delete it:

```
Command (m for help): d
Selected partition 1
Command (m for help): p
Disk /dev/sdb: 252 MB, 252968960 bytes
255 heads, 63 sectors/track, 30 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0067f7b9
Device Boot
Start
End
Blocks Id System
Command (m for help):
```

Lets create the FAT32 boot partition and mark it as bootable.

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

```
Command (m for help): n
Command action
e extended
p primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-30, default 1): 1
Last cylinder, +cylinders or +sizeK,M,G (1-30, default 30): +64M
Command (m for help): a
Partition number (1-4): 1
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
```

We will then create the ext3 partition that will host the root file system.

```
Command (m for help): n
Command action
e extended
p primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (10-30, default 10): Enter
Using default value 10
Last cylinder, +cylinders or +sizeK,M,G (10-30, default 30):
Enter
Using default value 30
Command (m for help): p
Disk /dev/sdb: 252 MB, 252968960 bytes
255 heads, 63 sectors/track, 30 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0067f7b9
Device Boot
/dev/sdb1 *
/dev/sdb2
Start
End
Blocks Id System
1 9 72261
c W95 FAT32 (LBA)
10 30 168682+ 83 Linux
```

Finally write the partition table and exit:

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

```
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
```

Format the partitions.

```
$ sudo mkfs.msdos -F 32 /dev/sdb1 -n BOOT
$ sudo mkfs.ext3 -L ROOTFS /dev/sdb2
```

Copy the bootloaders and kernel image onto the boot partition.

```
$ sudo mkdir /media/BOOT
$ sudo mount /dev/sdb1 /media/BOOT
$ sudo cp ~/oe-build/tmp/deploy/eglibc/images/
omap4430-panda/MLO-omap4430-panda /media/BOOT/MLO
$ sudo cp ~/oe-build/tmp/deploy/eglibc/images/
omap4430-panda/u-boot-omap4430-panda.bin /media/BOOT/u-boot.bin
$ sudo cp ~/oe-build/tmp/deploy/eglibc/images/
omap4430-panda/uImage-omap4430-panda.bin /media/BOOT/uImage
$ sudo umount /media/BOOT
$ sudo rmdir /media/BOOT
```

Extract the root filesystem to the ROOTFS partition.

```
$ sudo mkdir /media/ROOTFS
$ sudo mount /dev/sdb2 /media/ROOTFS
$ sudo rm /media/ROOTFS/* -r
$ sudo mkdir /media/ROOTFS/boot
$ sudo tar xjvf ~/oe-build/tmp/deploy/eglibc/images/
omap4430-panda/minimalist-image-omap4430-panda.tar.bz2
-C /media/ROOTFS
$ sudo umount /dev/sdb2
$ sudo rmdir /media/ROOTFS
```

On the development PC, open a terminal window using the minicom tool and configure it as follows :

```
$ sudo minicom -s
```

Select Serial Port Setup and change the following parameters if needed:

- Serial Device : /dev/ttyS0 (or else depending on the machine)
- Lockfile location : /var/lock
- Callin Program : (leave empty)

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

- Callout Program : (leave empty)
- Bps/Par/bits : 115200 8N1
- Hardware Flow Control : No
- Software Flow Control : No

Press Escape to exit then choose Save setup as dfl to save your parameters. Select Exit to start using the terminal window. You are now all set to start booting the PandaBoard. Plug the prepared SDCard into the boards slot, connect the power supply and the serial cable to the PC. And power on the board. You should see the Linux kernel boot in the console. Type `root` as login and you are now able to enter commands to the Linux shell.

If you would like to configure the network, use the following commands:

```
> ifconfig eth0 up
> udhcpc -i eth0
```

Getting the kernel sources

Create a working directory for compiling the kernel sources. We will use OpenEmbedded to automatically download and patch the kernel sources for the Panda Board. We will then copy the kernel source folder into the working directory.

```
$ mkdir -p ~/labs/lab1
$ cd ~/oe-build
$ export PATH=$PATH:~/bitbake-1.10.2/bin
$ export BBPATH=~/oe-build:~/openembedded
$ bitbake virtual/kernel -f -c patch
$ cd tmp/work/omap4430-panda-angstrom-linux-gnueabi
$ cp -r linux-omap4-2.6.35.3-r101c/git/
~/labs/lab1/linux-omap4-2.6.35.3
$ cd ~/oe-build
$ bitbake virtual/kernel -f -c clean
```

Cross-compiling environment setup

We first need to set up the cross-compilation toolchain. The toolchain is a set of tools including an ARM compiler that will run on an x86 host to build binaries targeted to an ARM platform. We will use the toolchain that has been generated by OpenEmbedded. To be able to call this toolchain from any path in the filesystem, we need to add the toolchain's directory into the PATH environement variable. We also need to set the ARCH and CROSS_COMPILE environment variables for the kernel build system to select the right architecture and toolchain to use for cross-compilation.

```
$ export PATH=$PATH:~/oe-build/tmp/sysroots/
x86_64-linux/usr/armv7a/bin
$ export CROSS_COMPILE=arm-angstrom-linux-gnueabi-
$ export ARCH=arm
```

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Linux kernel configuration

Pick up the default configuration for the Panda board.

```
$ cd ~/labs/lab1/linux-omap4-2.6.35.3
$ make <your_board_defconfig>
```

You don't know the name of the defconfig file for your board?

Take a look under the `arch/<ARCH>/configs` directory for the list of possible configurations. Launch a configuration tool to browse for the available configuration parameters. Look for the `local version` parameter and set a custom string that will be appended to the kernel version string.

Cross compiling

Launch the kernel compilation.

```
$ make uImage -j3 && make modules
```

Note: The `-j X` parameter defines the number of threads that will be created to perform the compilation. It is always interesting to use it on multi-core systems, general rule is to set it to the number of cores of the build machine plus one.

Booting your kernel

You should already have a running Linux image with a proper root filesystem on your SDcard. Follow the steps you have already done in the first part of this lab to update the kernel image on the SDcard. Make sure you use the kernel you just built!

If the boot process goes through and you reach a command line shell, congratulations! To make sure it is the right kernel, use the `uname` command to check the booted kernel's local version.

NFS booting

When developing under Linux, we often need to rebuild our module or application. To ease up re-deploying the newly generated binary to the target platform, it is very convenient to use a remote filesystem on the PC as the root filesystem of the platform. This can be done using a Network File System (NFS) that is physically located on the development machine, and to which the target platform connects to access it as its root filesystem. This way a program compiled from the NFS on the developement PC is directly available from the target!

To setup a NFS server, first edit the `/etc/exports` file as root on the development PC to add the following entry.

```
/srv/nfs/rootfs *(rw,sync,insecure,no_root_squash,no_subtree_check)
```

Unpack the root file system somwehere on your development machine and create a link to this directory.

```
$ mkdir ~/labs/lab1/rootfs
$ sudo tar xjvf ~/oe-build/tmp/deploy/eglibc/images/
omap4430-panda/minimalist-image-omap4430-panda.tar.bz2 -C
~/labs/lab1/rootfs
$ sudo mkdir -p /srv/nfs
$ sudo ln -s ~/labs/lab1/rootfs /srv/nfs/rootfs
```

Then restart the NFS server.

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

You then need to change U-BOOT's parameters to set the right kernel command-line that will allow the kernel to boot the NFS root file system instead of the one located on the SDcard. We will create a U-BOOT script that will be automatically loaded at startup to set the right variables. Edit a file named `boot.script` and copy the following content:

```
setenv nfsargs 'setenv bootargs console=${console} vram={vram}
        root=/dev/nfs nfsroot=${serverip}:/srv/nfs/rootfs rw ip=dhcp'
setenv serverip <ip_address_of_the_pc>
setenv mmcboot 'echo Booting from mmc${mmcdev} ...; run nfsargs;
        bootm ${loadaddr}'
run loaduimage
run mmcboot
```

This script needs to be converted for U-BOOT to load it. Use the `mkimage` tool as follows to generate a proper U-BOOT script.

```
$ mkimage -A arm -T script -C none -n "Bootscript" -d
boot.script boot.scr
```

Copy the `boot.scr` file onto the `BOOT` partition of the SDcard, insert it into the PandaBoard and boot it up. As the root filesystem is now located on a remote PC on the network, make sure you plugged an ethernet cable between your board and the network!

Installing the modules

In the previous steps, you rebuilt the kernel and modules but only the kernel was actually deployed to the board. As a kernel can only load modules generated from the same build, you can no longer load the modules that are currently part of the root file system. You hence need to deploy the freshly built modules to your root file system.

```
$ sudo make modules_install ARCH=arm
INSTALL_MOD_PATH=<installation_path>
```

Be careful, if you don't specify the INSTALL_MOD_PATH variable, the build system will install the modules to `/lib/modules` by default, hence installing modules built for an ARM architecture to the x86 root filesystem of your development machine.

To make sure you correctly deployed the modules to your root file system, boot up the platform and try loading a module with the `modprobe` command.

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Lab 2 : Using OpenEmbededd Tools

Objective: Learn how to use the OpenEmbedded tools to generate a Linux image and add new packages to the distribution.

After this lab you will be able to:

- Use the bitbake tool to build an image for the PandaBoard
- Deploy the generated binaries onto the target
- Write a simple Linux application
- Create a recipe for your application
- Use OpenEmbedded to build your application
- Install a package from a remote repository

Build an image using OpenEmbedded

OpenEmbedded is made of configuration files that are parsed to generate binaries that will be later deployed onto the final root filesystem. The OpenEmbedded distribution is located at `~/openembedded` on your developement machine.

Have a look at the files located there to get familiar with the concept of ”recipes”. Open one recipe and try to understand how it works. We will now launch a full build of the distribution for our target platform.

First go to the build directory that holds the build result. It also contains temporary files and source code as the build goes.

```
$ cd ~/oe-build
```

You need to prepare a configuration file to define the board you want to target, and configure some parameters for the build. A sample configuration file is provided along with OpenEmbedded, just copy it to the right directory before editing it.

```
$ cp ~/openembedded/conf/local.conf.sample conf/local.conf
```

Edit the `local.conf` file and fill up the following parameters:

- BBFILES: Set ”/home/trainee/openembedded/recipes/*/*.bb” to specify the path to the recipes
- MACHINE: Select the Panda Board as the target machine. To figure out the exact name of the machine, have a look under the available machines under the `~/openembedded/conf/machine/` directory
- DISTRO: Set ”angstrom-2010.x” to select the distribution to compile. Look under `~/openembedded/conf/distro` for a complete list of available distros.
- PARALLEL_MAKE: Set ”-j 3” to parallelize compilation across 3 different threads
- BB_NUMBER_THREADS: Set 2 to parallelize the build process and speed up the build

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

- INHERIT += "rm_work": Set this variable to delete temporary files along the build and save up a lot of space
- REMOVE_THIS_LINE: As specified, delete this line to allow the build to launch

Set the proper environment variables before launching the build:

```
$ export PATH=$PATH:~/bitbake-1.10.2/bin
$ export BBPATH=~/openembedded:~/oe-build
```

Launch a build for generating a minimal root filesystem for your board.

```
$ bitbake minimal-image
```

Wait for the build to finish. Note that for the purpose of this lab the packages were prebuilt. Doing such a build from scratch usually takes several hours to complete!

Deploy the generated image to the target

Follow the steps detailed in lab 1 to deply this newly generated image onto your board. Make sure you update the bootloader and kernel images, then extract the generated root file system into a new directory named `~/labs/lab2/rootfs`. Make sure you change the `/srv/nfs/rootfs` symbolic link to point to the location of this new root file system. Do not forget to restart the NFS server to take your new root file system into account.

Write a simple application

Create a new directory named `~/labs/lab2/rootfs/src` to store your application. This directory is located under the root file system which is owned by `root`, you hence need to change this directory permissions to allow creating files and compiling in this location.

```
$ sudo chown trainee ~/labs/lab2/rootfs/src
$ sudo chgrp trainee ~/labs/lab2/rootfs/src
```

Create a simple C program under this directory with a `main` function as follows:

```
int main(int argc, char *argv[])
{
    return 0;
}
```

Fill up the main function to add code that retrieves the version string of the running kernel and display it on the standard output.

Write a `Makefile` for cross-compiling your application using the sample below:

```
CC = $(CROSS_COMPILE)gcc

default:
    $(CC) -o <executable_name> <c_file> ${LDFLAGS}

clean:
    rm -rf *.o *.swp
```

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Don't forget that Makefiles requires tabs for indentation and not spaces!

Build your application and deploy the resulting binary onto the target. Try out your application and check you get the correct version string for your kernel.

Create an OpenEmbedded recipe for your package

Under `~/openembedded/recipes`, create a directory to hold your application package. Create a new recipe file for your application in it with the following naming convention: `<package-name>_<version-number>.bb`. You also need to create a subdirectory inside your recipes's directory to keep a tarball of your application's sources. The naming convention of this subdirectory should be as follows: `<package-name>-<version-number>`. Pack up your application's sources into a ".tar.gz" file and copy it under this newly created subdirectory. Finally edit your recipe (.bb) file and fill it up as done in the example below.

```
DESCRIPTION = "<short_description_of_your_application>"
LICENSE = "GPLv2"

SRC_URI = "file://<package_name>-${PV}.tar.gz"

S = "${WORKDIR}/<package_name>"

inherit autotools

EXTRA_OEMAKE = "CROSS_COMPILE=${TARGET_PREFIX}"

do_install() {
    install -d ${D}${bindir}
    install -m 755 <executable_name> ${D}${bindir}
}
```

Building the package

In this next step, use the bitbake tool to build your custom package and create an ".ipk" package. Such packages can then be easily deployed into the target root filesystem.

```
$ cd ~/oe-build
$ export PATH=$PATH:~/bitbake-1.10.2/bin
$ export BBPATH=~/openembedded:~/oe-build
$ bitbake <package_name>
```

When the build process is over, you should find your package's .ipk file under:

`~/oe-build/tmp/deploy/eglibc/ipk/armv7a`

The final step to make this package available is to reconstruct the package index. Launch the specific target as follows to expose your new package.

```
$ bitbake package-index
```

We will then learn how to install this package over the network from a remote repository located onto your development machine.

Deploying an OpenEmbedded package

We first need to launch a web server on the development PC to allow the target to download packages. We will use busybox on the development machine which contains a simple web server that is very easy to configure.

```
$ sudo apt-get install busybox
$ sudo busybox httpd -h ~/oe-build/tmp/deploy/eglibc/ipk
```

Edit `/etc/opkg/omap4430-panda-feed.conf` on the target platform and add the following lines.

```
src/gz armv7a http://<ip_of_the_dev_pc>/armv7a
src/gz omap4430-panda http://<ip_of_the_dev_pc>/omap4430-panda
src/gz all http://<ip_of_the_dev_pc>/all
```

Make sure the network is properly configured between the target and the development PC and run the following command on the target.

```
opkg update
```

Finally install the package hosted by the remote PC to the target with the following command.

```
opkg install <package-name>
```

If everything went fine, your package should install and your application should be available under `/usr/bin`. Try to launch your application and see if it works as expected.

Please note that you can install any package that has been compiled by OpenEmbedded. Try out a few packages on your board by installing them with the `opkg` command.

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Lab 3 : Deploying an Android image and debug a JAVA application

Objective: Learn how to deploy a prebuilt Android image on the target and deploy/debug a JAVA application over ADB USB.

After this lab you will be able to:

- Prepare a SDcard with an Android prebuilt image
- Run the Android image on the target platform
- Write a JAVA application using Eclipse IDE and deploy it on the target using ADB
- Debug the application from the IDE

Note: The process you will follow during this lab is documented on http://www.linaro.org, from which the prebuilt image has been downloaded. Feel free to look for resources on this website.

Deploy an Android image

For the purpose of this lab, we will use a prebuilt Android image downloaded from http://www.linaro.org. First uncompress the image.

```
$ cd ~/labs/lab3
$ tar xJvf linaro-image-lab3.xz
$ cd linaro-image
```

Insert an SDcard into the reader and figure out the name of the device (e.g. /dev/sdX). Use Linaro tools to prepare the SDcard using the prebuilt packages. We also need to call a script that will patch the image with specific binaries for enabling graphics hardware acceleration.

```
$ sudo umount /dev/sdX*
$ sudo linaro-android-media-create --mmc /dev/sdX --dev panda
   --boot boot.tar.bz2 --system system.tar.bz2 --userdata
   userdata.tar.bz2
$ ./install-binaries-4.0.4.sh
```

Accept the license when prompted. When the script ends up successfully, properly unmount the SDcard.

```
$ sudo umount /dev/sdX*
```

Eject the SDcard and plug it into the Pandaboard. Power up the board and wait until you reach the Android launcher. Start playing around in Android to check the available applications.

The Android port for the Pandaboard does not support the suspend and resume features. You need to disable this feature using the `DisableSuspend` application to avoid the board being

freezed after the suspend timeout.

Setup the ADB connection over USB

While Android is running on your platform, connect a USB cable between the board Mini-USB port and the development PC. Start up the ADB server on the PC with root permissions.

```
$ cd ~/android-sdk-linux/platform-tools
$ sudo ./adb kill-server
$ sudo ./adb start-server
$ ./adb devices
List of devices attached
0123456789ABCDEF device
```

You should see an attached device listed as the example below. This means that the platform is properly connected over ADB, and you can start developing and deploying applications from the Eclipse IDE.

Setup Eclipse to work with the Android SDK

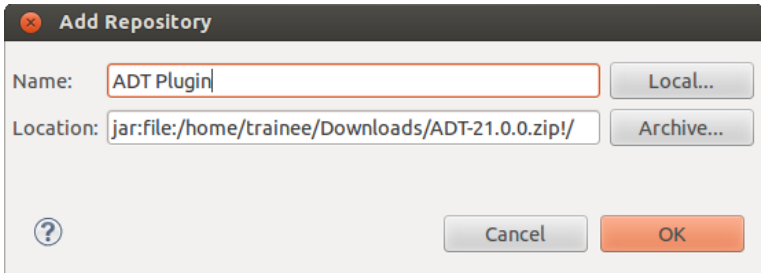
You first need to download the ADT plugin to install into Eclipse.

```
$ cd ~/Downloads
$ wget http://dl.google.com/android/ADT-21.0.0.zip
```

Launch Eclipse from the command line shell.

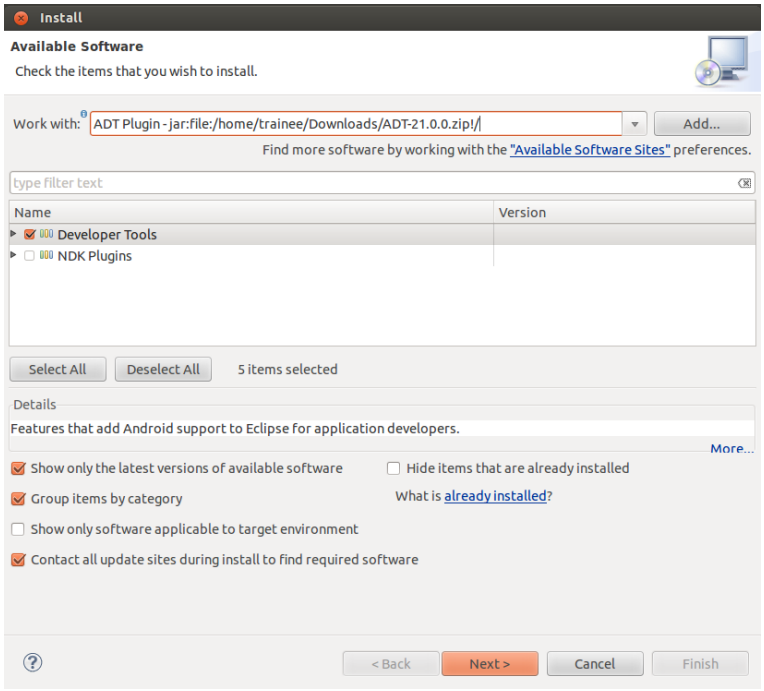
```
$ ~/eclipse/eclipse &
```

If prompted, select a directory where to store your workspace that will contain your JAVA applications. Once you reached the Eclipse interface, click the `Help` menu then click `Install new sotware`. In the dialog box, click the `Add` button in the upper right corner. Click the `Archive` button and browse for the ADT plugin under `~/Downloads/ADT-21.0.0.zip`.

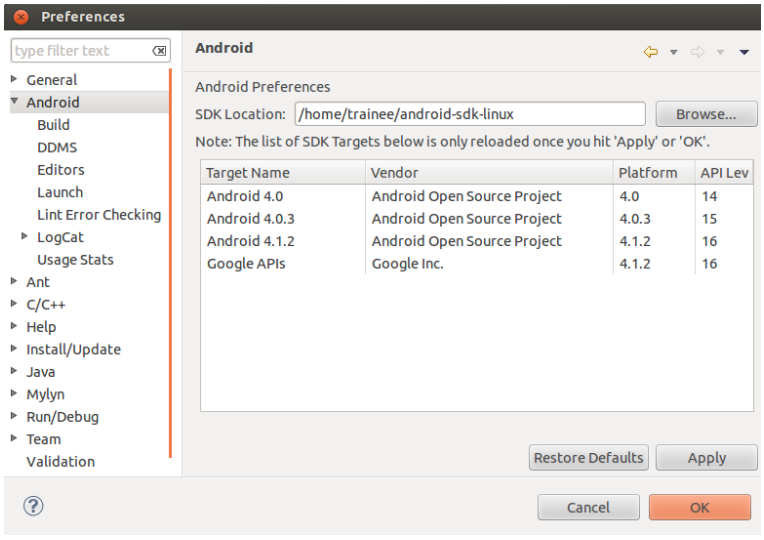


In the nex dialog box, select `Developer Tools` and click `next`.

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

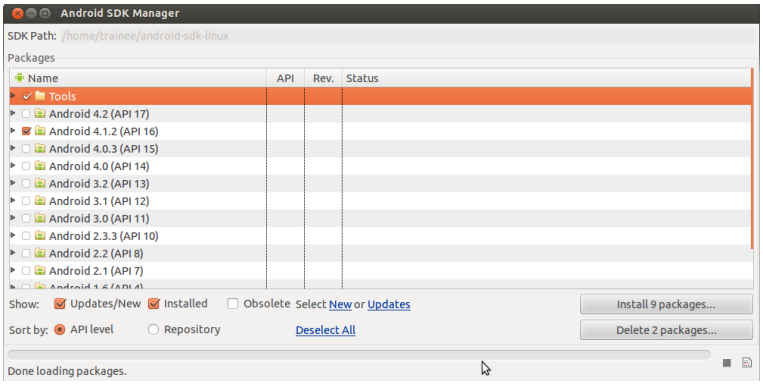


Wait for the tools to be downloaded and installed, then Eclipse should restart. If not automatically redirected, open Eclipse preferences (`Window -> Preferences`) and select the `Android` item. Change the SDK location to point to `/home/trainee/android-sdk-linux` then click Ok.



In the `Android SDK Manager`, select `Tools` and `Android 4.1.2 (API 16)` for installation.

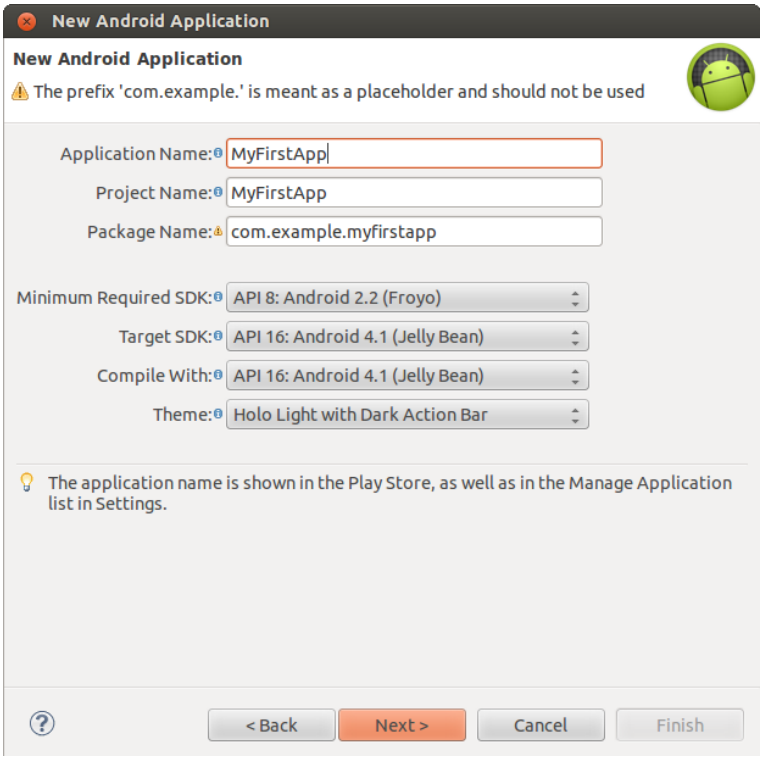
Note: To save up in download time and installation space, you can unselect the `Intel x86 Atom System Image` and `MIPS System Image` items from the `Android 4.1.2 (API 16)` subcategory.



Click `Install` to finalize the installation of the SDK.

Deploy a simple Java application over ADB

Under Eclipse, Click `File -> New -> Project` to open the new project dialog. Select `Android Application Project` in the next dialog. Enter any name (e.g. `MyFirstApp`) under the `Application Name` text entry. Make sure you select `API 16: Android 4.1 (Jelly Bean)` as the `Target SDK`. `Minimum Required SDK` can remain set to an older SDK such as `API 8: Android 2.2 (Froyo)`.



Click `Next` several times until you reach the last dialog, then click `Finish`. Eclipse should generate a simple "HelloWorld" project that you can modify. Start by adding a button to your application by opening `res/layout/activity_main.xml`. In the Graphical Layout view, drag and drop a button from the left panel to the center of your applcation's activity.

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

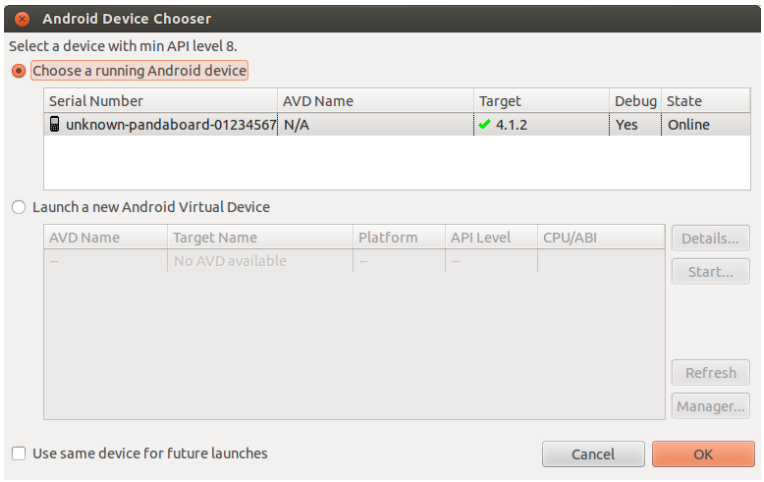
Now edit `MainActivity.java` under the `src` folder to add a listener function to the newly created button. At the end of the `OnCreate` function, add the following code.

```
Button button1 = (Button)findViewById(R.id.button1);
button1.setOnClickListener(new OnClickListener() {
        public void onClick(View view) {
                Log.i("MyFirstApp", "Button 1 has been pressed");
        }
});
```

Note: You will most likely need to add imports before your project properly compiles. Place the mouse cursor on top of the red underlined portions of code, a context menu will appear providing a shortcut for adding the proper import declaration.

Finally, add a breakpoint to your button listener function by placing the mouse cursor on the line inside the `onClick` function, then clicking `Run -> Toggle Breakpoint`. It should place a small blue dot on the left side of the editor window.

Click `Run -> Debug` to start deploying and debugging your application. When the `Android Device Chooser` dialog pops up, select the attached Pandaboard then click Ok.



Eclipse should deploy the application on the board, then switch to the Debug perspective. After a few seconds, the application should appear on the Pandaboard's display. Click the button on the screen, the debugger should stop the program execution and hit on the breakpoint you set. Press F6 to execute the current instruction, observe that the information trace is correctly displayed under the `Logcat` window.

That's it, you deployed a Java application onto the target and learned how to debug it. Play around with the SDK and try adding new features to your application.

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Lab 4 : Write an image converter application and optimize it using native code

Objective: Learn how to optimize algorithms using native libraries

After this lab you will be able to:

- Write an application to convert an image and display the result
- Measure time needed to perform the conversion
- Use the NDK to write a library that implements the algorithm in native code
- Call the native library from the JAVA application and compare the computation time results

Write a JAVA application to convert an image

The goal of this application is to convert an image whose pixels are stored under a ABGR888 color format into ARGB888. The image to be converted is named `LinuxAdeneoBGR.jpg`. This image needs to be copied to the Android device into its external storage (using the `adb push` command for instance). Then the application should load this image, perform the color conversion and finally display the result.

The final application should look like this:

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

All the information you need to write such an application is available from Google's Android reference website. However to help you get started, here are a few hints.

Find the device's external storage path: The external storage path is different from one Android device to an other. Instead of using hard-coded path in your application, it is better to call this dedicated API:

```
File sdDir = Environment.getExternalStorageDirectory();
```

Load an image and copy data into a buffer: To access picture data at the byte level, we need to load the picture and copy its pixels into a `Buffer` object whose bytes may be directly accessed. The following code snippet loads an image from the file system and loads its data into a `ByteBuffer` object:

```
Bitmap Bmp = BitmapFactory.decodeFile(strFileName);
if (Bmp == null)
{
        Log.i("FAILED:", "Could not find " + strFileName);
        return;
}

ByteBuffer Buf = ByteBuffer.allocate(Bmp.getHeight()*Bmp.getWidth()*4);
Bmp.copyPixelsToBuffer(Buf);
```

Measure computation time: In order to measure the time elapsed while performing a computation, we can rely on the system timer as follows:

```
long startTime, duration;

startTime = System.currentTimeMillis();
// Computation code to be measured
duration = System.currentTimeMillis() - startTime;

Log.i("INFO:", "Duration: " + Long.toString(duration) + " ms");
```

Copy a Bitmap object: In order to generate a new bitmap for holding the result of the conversion, we can copy the source bitmap into a new `Bitmap` object with the same characteristics:

```
Bitmap dstBmp = srcBmp.copy(Config.ARGB_8888, true);
```

Create a device-independent graphical layout: To make sure your graphical interface will be displayed properly on devices that have different screen resolutions, you should use relative layouts to arrange your UI elements. The best way is to use horizontal and vertical `LinearLayout` objects. Following is a sample layout XML file that uses such objects to build up the graphical interface shown in the picture above.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="fill_parent"
```

```
            android:layout_weight="90" >

            <ImageView
                android:id="@+id/imageViewSrc"
                android:layout_width="fill_parent"
                android:layout_height="match_parent"
                android:layout_weight="50" />

            <ImageView
                android:id="@+id/imageViewDst"
                android:layout_width="fill_parent"
                android:layout_height="match_parent"
                android:layout_weight="50" />
        </LinearLayout>
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

            <TextView
                android:id="@+id/textViewDuration"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:text="Duration: 0ms"
                android:textAppearance="?android:attr/textAppearanceLarge" />
        </LinearLayout>
    </LinearLayout>
</RelativeLayout>
```

Write a native library using the NDK

For optimization purposes, we will now move the color conversion algorithm to native code by implementing it in C code. This code will be directly executed from the CPU, which is a lot faster than performing JAVA operations that will have to be interpreted by the Dalvik virtual machine.

First download the NDK from Google's website and extract it:

```
$ cd ~/Downloads
$ wget http://dl.google.com/android/ndk/
      android-ndk-r8c-linux-x86.tar.bz2
$ cd ~
$ tar xjvf Downloads/android-ndk-r8c-linux-x86.tar.bz2
```

We will now need to create a JNI interace to have the native library be called from JAVA code. Create a new JAVA class in your Eclipse project, and edit the corresponding JAVA class to export a function as follows:

```
package com.example.bitmapconvert;

public class NativePictureConverter {
    static {
            System.loadLibrary("NativePictureConverter");
        }

    /**
```

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

```
     * Convert a byte buffer from BGR format to RGB
     */
    public native void ConvertBGRtoRGB( byte[] pSrc, byte[] pDst, int width, int height);
}
```

Next step is to create a JNI header out of this newly created class. Under a terminal window, change directory up to your workspace's classes directory. Then call the `javah` tool to automatically generate the header file. Finally move this file to a newly created `jni` folder.

```
$ cd <workspace>/<myapp>/bin/classes
$ javah -jni com.example.bitmapconvert.NativePictureConverter
$ mkdir ../../jni
$ mv com_example_bitmapconvert_NativePictureConverter.h ../../jni
```

Create a new C file under `<workspace>/<myapp>/jni` that will hold your native code. In this file, include the JNI header file that you just generated. Implement the functions whose prototypes are defined in this header. Do not forget to enter parameters name in the function declaration if you pasted the prototype from the header file.

Under the `ConvertBGRtoRGB` function, implement the algorithm that will perform the BGR to RGB conversion.

Note: `jbyteArray` objects cannot be accessed directly. You first need to retrieve a proper pointer to the actual byte array as follows:

```
jbyte *c_src = (*env)->GetByteArrayElements(env, pSrc, 0);
```

Once the algorithm is properly implemented, create a `Android.mk` file under the same directory to allow proper compilation of the library by the Android build system. Copy the following content into this file.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := <Name of the library>
LOCAL_SRC_FILES := <file_to_compile>.c

include $(BUILD_SHARED_LIBRARY)
```

You can now build your native library using the NDK.

```
$ cd <workspace>/<myapp>/jni
$ export PATH=$PATH:~/android-ndk-r8c
$ ndk-build
```

Make sure the library compiled properly. Refresh your project under Eclipse and check that your newly built library is properly displayed under the `libs` directory. Now that your native library is included into your JAVA project, you can call it from your JAVA code as follows.

```
NativePictureConverter nPicConv = new NativePictureConverter();
nPicConv.ConvertBGRtoRGB(...);
```

Replace your conversion algorithm written in JAVA by a call to the native conversion function and run your application on the target. Compare the results and see how much you improved the performance of the conversion.

Optimizing further

The result may be impressive, there is a way of optimizing the algorithm even more! What is very time consuming in this algorithm is the fact that a lot of iterations (equals to the number of pixels) are needed to perform the full conversion, and each of those iterations calls many processor instructions to do the swapping. The current solution may be suitable for most needs, it may not be enough in case you have to deal with big resolution pictures (more iterations) or with many pictures in a limited time frame (video stream at 60 fps).

Luckily, the ARM Cortex A9 architecture includes a NEON coprocessor that can help in your case. This coprocessor is based on a SIMD architecture which allows performing operations on up to 8 vectors of 4 bytes simultaneously while incrementing and index.

There are two ways of calling NEON instructions from an application:

- Intrinsics: Those are C functions you can directly call from C code. They are a quick way of using the NEON without the need for writing assembly language. Also note that some NEON instructions do not have an intrinsic equivalent.
- Assembly: NEON instructions can be directly called from assembly language. This is the most performant way of using NEON, this is also the most complicated one. If you decide to go that way, GCC inline assembly may be a good solution.

Note: For the NDK to compile NEON instructions, you need to set some compilation flags in `Android.mk`.

```
LOCAL_CFLAGS    := -march=armv7-a -mfloat-abi=softfp -mfpu=neon
```

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Lab 5 : Write a character driver and call it from an Android application

Objective: Learn how to write a simple driver that accesses the hardware and provides an interface to the applications.

After this lab you will be able to:

- Write a simple character driver
- Call the kernel GPIO library to toggle a LED
- Expose driver's functions to userspace
- Write an Android application that calls the driver to access the LED

Recompile the kernel for Android

We first need to recompile the kernel to have a base from which to link the module we will create. The kernel sources are located under the linaro source tree at `~/linaro-build/android/kernel`.

In order to have the touchscreen report the correct coordinates to Android, we first need to apply a patch that inverts the X and Y coordinates reported to Android. We will use git to easily apply this patch. Copy the patch named `Report-inverted-X-an-Y-coordinates.patch` under the `~/linaro-build/android/kernel` directory and type the following commands.

```
$ cd ~/linaro-build/android/kernel
$ git apply Report-inverted-X-an-Y-coordinates.patch
$ export ARCH=arm
$ export PATH=$PATH:~/linaro-build/android/prebuilts/gcc/
    linux-x86/arm/arm-linux-androideabi-4.6/bin
$ export CROSS_COMPILE=arm-linux-androideabi-
```

Now configure the kernel to be built for an OMAP4 Panda board. The corresponding `defconfig` file is named `android_omap4_defconfig`. Then open any configuration tool to add the `CONFIG_ARM_APPENDED_DTB` missing feature.

Note: This configuration option is used to have the `Device Tree Blob (dtb)` file appended to the kernel image. This way we do not have to manually load it before starting up the kernel.

Finally launch the kernel compilation.

Write a simple character driver

Create a new directory for holding your module's source code and change to it.

```
$ mkdir -p ~/labs/lab5/module
$ cd ~/labs/lab5/module
```

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Create a new `.c` file under this directory and add the following code skeleton.

```
nclude <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init led_init(void)
{
    return 0;
}
static void __exit led_exit(void)
{
}

module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("LED driver");
MODULE_AUTHOR("Your name");
```

This is the minimal code you need to provide to allow building this as a module. Now create a `Makefile` under this same directory and copy the following content.

```
obj-m := <name of your C file>.o
# KDIR should point the the sources of the kernel to build against
KDIR := /home/trainee/linaro-build/android/kernel

all:
        $(MAKE) CFLAGS_MODULE=-fno-pic -C $(KDIR) M=`pwd` modules
```

Try to build your module using the `make` command after setting the usual `PATH`, `ARCH`, and `CROSS_COMPILE` environment variables as you have done previously to build the kernel.

If everything goes well, you should end up with a generated `.ko` file. This is your module!

Deploy the kernel and module on the platform

It is time to check that the kernel and module you just compiled are working well on the platform. Insert the SDcard into the reader and open the FAT partition usually named `boot`. Replace the `uImage` file located on the SDcard by the one you just compiled located under `~/linaro-build/android/kernel/arch/arm/boot`. Plug back the SDcard into the PandaBoard's slot and let the platform boot. Android should normally start and the touchscreen should be functional.

We will then use `ADB` to copy our module on the target. This way we can easily redeploy it after modification.

```
$ export PATH=$PATH:~/android-sdk-linux/platform-tools
$ adb push <module name>.ko /data
```

Once the module has been properly deployed onto the target, you can try to load it into the kernel by typing the following commands on the target shell.

```
> insmod /data/<module name>.ko
```


© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

If no error was output on the terminal, then you module has most likely been loaded. To verify it is loaded, type the `lsmod` command and check that your module name is correctly displayed in the list.

Finally unload the module using the command below.

```
> rmmod <module name>
```

Modify the module to have it toggle a LED

Modify the simple driver you just wrote to add a feature that allows setting the state of the onboard LED labeled `D1`.

Note: LEDs are usually controlled by a "General Purpose Input Ouput" (GPIO) from the processor. The GPIO number corresponding to led `D1` can be found in the PandaBoard schematics by looking for the signal that goes from the LED to the CPU.

Following are a few hints to help you get started.

Use the GPIO kernel library: GPIOs can be easily controlled using the dedicated kernel SDK. Include `linux/gpio.h` into your driver and use the functions exposed by this header file such as the functions defined below.

```
int gpio_request(unsigned gpio, const char *label)
void gpio_free(unsigned gpio)
int gpio_direction_input(unsigned gpio)
int gpio_direction_output(unsigned gpio, int value)
int gpio_get_value(unsigned gpio)
void gpio_set_value(unsigned gpio, int value)
```

Implement LED control through the read and write functions: It is a convenient way to let the user controle the LED from an application. Implement the `write`, and `read` functions in your driver. In the `write` functions, parse the data that is passed by the user in the input buffer, if it is "0" turn the LED off and turn it on if "1" is passed instead. You can also have the `read` function return the current state of the LED ("0" or "1") in the output user buffer. Inspire from the example in the class materials to understand how to implement these functions and register your driver as a character device driver.

Test your driver using the associated node: Your driver should register as a character device driver with major and minor numbers. This is enough for you to create a node (using the `mknod` command) and start communicating with your driver as follows

```
> mknod /dev/<name of the driver node> c <major> <minor>
> echo 0 > /dev/<name of the driver node>
```

for sending string "0" as the input buffer of the `write` function of the driver.

```
> cat /dev/<name of the driver node>
```

for reading the ouput buffer returned by the `read` function of the driver.

© 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

Create an Android application that calls the driver

Now that you have a working driver, create a JAVA application whose goal is to turn the LED on and off. You can use for instance a `ToggleButton` object that will reflect the state of the LED and change its state when pushed. To achieve that, your application will need to call the driver using its node (/dev/xxxx) which can only be done from C/C++ code. You will hence need to create a JNI library to implement the LED driver access in C and make it available to the JAVA application. Refer to lab 4 for creating the JNI library if you forgot how to do it!

You should already know everything to create this application, but here are a few hints you can use to get started.

Use POSIX APIs to open and call the driver: The POSIX API used to communicate with drivers is the same that is used to open and handle files. Use `man` pages using the commands below to understand how these functions work and what parameters they expect.

```
$ man 2 open
$ man 2 write
$ man 2 read
$ man 2 ioctl
$ man 2 close
```

Check for proper permissions: By default, the node you created with `mknod` may only have permissions set for root access. This can be a problem as your application will run as a specific user that does not have full privileges. To have your application be able to access the driver, you will first have to change the permissions of the device node as follows.

```
$ chmod 0666 /dev/<name of the driver node>
```

Have your driver automatically load at startup: So far you have been loading your driver manually with the `insmod` command. If you reboot your platform, your application will no longer be able to access your driver until you manually reload it. To have Android automatically load your module at startup, you need to modify the `init.rc` file. This is located into a ramdisk file stored in the boot partition. Follow these steps to mount the ramdisk, modify it and regenerate it for update on the SD card.

```
$ dd bs=1 skip=64 if=/media/boot/uInitrd of=initrd.gz
$ gunzip initrd.gz
$ mkdir fs
$ cd fs
$ cpio -id < ../initrd
```

Under the `fs` directory you now have the ramdisk uncompressed. Modify the `init.rc` file and add the following lines under the `on boot` section.

```
insmod /system/modules/<name_of_your_module.ko>
chmod 0666 /dev/<name_of_the_driver_node>
```

Regenerate and update the ramdisk with the following commands.

 © 2004-2012 Free Electrons, Adeneo Embedded, CC BY-SA license

```
$ find ./ | cpio -H newc -o > ../newinitrd
$ cd ..
$ gzip newinitrd
$ mkimage -A arm -O linux -C gzip -T ramdisk -n "My Android Ramdisk
    Image" -d newinitrd.gz uInitrd-new
$ sudo cp uInitrd-new /media/boot/uInitrd
```

Finally make sure you copy your module to `/system/modules` on the system partition. Plug the SD card into the PandaBoard's slot and power up the board. Check that your driver has been properly loaded at startup with the `lsmod` command.