# Android on BeagleBone Black

**UTN-FRA**
**Laboratorio de Software Libre**

**Ernesto Gigliotti**

## Contents



**Examples: https://sourceforge.net/projects/androidonbeaglebonebtutorials**

| Rev. | Corrections |
|---|---|
| - | Original document |
| 1 | SPI Tutorial: clock pin must be input |
| 2 | Typographical errors, Kernel 3.8, PWM,ADC |

## License



**“Android on Beaglebone Black” by Ernesto Gigliotti.**
**Contact : ernestogigliotti@gmail.com**

**Tutorial 01:** *Creating SD card Android Texas Instruments Image for Beaglebone Black.*

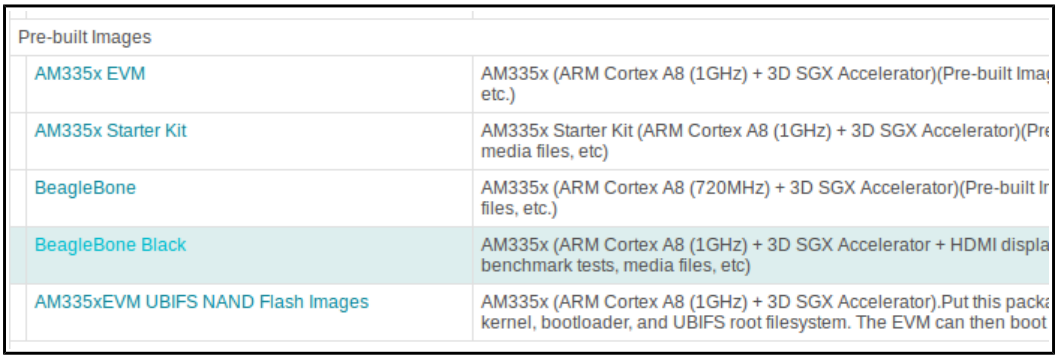
*Requirements:*

- This tutorial is based in a PC with Ubuntu 13 64bit Operating System.
- Micro SD card class 10 preferentially.

First of all, we have to download the TI Android Image from TI's web page:

http://downloads.ti.com/sitara_android/esd/TI_Android_DevKit/TI_Android_JB_4_2_2_DevKit_4_1_1/index_FDS.html

In the "Pre-Built Images" section, we download "Beaglebone Black"



| Pre-built Images | |
|---|---|
| AM335x EVM | AM335x (ARM Cortex A8 (1GHz) + 3D SGX Accelerator)(Pre-built Ima etc.) |
| AM335x Starter Kit | AM335x Starter Kit (ARM Cortex A8 (1GHz) + 3D SGX Accelerator)(Pre media files, etc) |
| BeagleBone | AM335x (ARM Cortex A8 (720MHz) + 3D SGX Accelerator)(Pre-built Ir files, etc.) |
| BeagleBone Black | AM335x (ARM Cortex A8 (1GHz) + 3D SGX Accelerator + HDMI displa benchmark tests, media files, etc) |
| AM335xEVM UBIFS NAND Flash Images | AM335x (ARM Cortex A8 (1GHz) + 3D SGX Accelerator).Put this packa kernel, bootloader, and UBIFS root filesystem. The EVM can then boot |

When the file is finished to download, we have to uncompress it. Inside the folder we will find the script "mkmmc-android.sh" with this script we will be able to create an Android Image for our Beaglebone Black.

*Steps*

1) Plug your micro SD card in the SDCard reader.

If you have a notebook with an integrated reader, the card will appear in /dev/mmcblk. If you have an USB reader, the card will appear in /dev/sdc or /dev/sdX it will depend how many partitions you have in your computer.

In this tutorial we will use mmcblk device

2) As root, grant execution permissions for the script:

```
chmod +x  mkmmc-android.sh
```

3) Execute the script and image will be copied into the SD card:

```
./mkmmc-android.sh /dev/mmcblk
```

**WARNING:** If you put a wrong path where it says "mmcblk"you can destroy your partition.

Once the script finished, you can put the SD card into the Beaglebone Black and power it, Android will boot.

**Tutorial 02:** *Accessing the Android Console by ADB*

*Requirements:*

- Android SDK installed in your computer

When we plug the Beagle's mini USB connector to our computer, we can connect to Android by ADB.

The first step is to configure Ubuntu for recognize the Beaglebone Black as a valid USB device.

*Steps*

1) As root, create the file "/etc/udev/rules.d/51-android.rules "

```
nano /etc/udev/rules.d/51-android.rules
```

2) Put the following text inside the file:

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="0451", MODE="0666"
```

3) As root, change the file's permissions:

```
chmod a+r /etc/udev/rules.d/51-android.rules
```

4) As root, go to the adb directory inside Android-SDK and kill adb and restart it listing the devices detected (Beaglebone Black must be plugged):

```
./adb kill-server
./adb devices
```

The output will be something like:

```
List of devices attached
0123456789ABCDEF  device
```

5) Then we are able to enter in the Android console typing:

```
./adb shell
```

6) if we want to copy a file from our computer to the Beaglebone Black we can use the "push" adb's command:

```
./adb push ~/beagleboneblack/foo.txt /etc/
```

We can too open Eclipse and use the board as another emulator or phone, downloading applications in it.

**Tutorial 03:** *Controlling GPIOs over Android Operating System*

In this tutorial we are going to control user's led in the board.

Processor pins for Users Leds:

| | |
|---|---|
| USR0 | GPIO1_21 |
| USR1 | GPIO1_22 |
| USR2 | GPIO1_23 |
| USR3 | GPIO1_24 |

GPIO control is not available for default, we need to create a directory for each GPIO we want to control, and there will be files in there that will allow us to control the pins.

*Steps*

1) Enter to Android Console by adb

```
./adb shell
```

2) Go to directory sys/class/gpio:

```
cd /sys/class/gpio
```

3) There, we will find the files and directories:

```
export
gpiochip0
gpiochip32
gpiochip64
gpiochip96
unexport
```

Each gpiochipXX corresponds to a 32 bit GPIO Port. For pin number calculation, we need to calculate the follow:

pin number = (port number * 32) + gpio number

for example, for user led 0 ( GPIO1_21 )

pin number = (1 * 32) + 21 = 53

3) Now we know the pin number we want to control, we need to generate the directory for this pin, that is made writing the pin number in the “export” file:

```
echo 53 > /sys/class/gpio/export
```

4) Done that, a new directory will appear:

```
export
gpio53       <- This directory is new
gpiochip0
gpiochip32
gpiochip64
```

```
gpiochip96
unexport
```

5) Go to this directory:

```
cd gpio53
```

6) There, we will find some files:

```
active_low
direction
edge
power
subsystem
uevent
value
```

7) We will use “direction” file to configure the pin as input or output. We write “high” in it for output:

```
echo high > direction
```

8) Now we can control the pin state using “value” file, writing a “1”, user led 0 will turn on, and writing a “0”, user led 0 will turn off.

```
echo 0 > value
echo 1 > value
```

*Considerations:*

When we reboot the board the “gpio53” directory will not be available and we will need to create it again. This problem is saved using a boot script where this initialization is made.

**Tutorial 04:** *Creating a boot script for hardware initialization.*

If we want that GPIOs (or other hardware modules) are initialized when Android boots, we need to create a script with the gpios configuration steps. We will name this file "**scriptgpios.sh**"

*The content of this file will be:*

```
#Configure LED USER 0 - GPIO1_21
echo 53 > /sys/class/gpio/export
cd /sys/class/gpio/gpio53
# Direction : out
echo high > direction
echo 0 > value
chmod 777 value
chmod 777 direction
```

Now we need to execute this script at the beginning of the Android booting. For that, we need to edit the "**init.rc**" file that is placed in Android's file system root.

Since we can not edit the file inside Android Operating System (we do not have a text editor in Android console) we are going to edit the file plugging the SD card in our computer and editing the file from there.

When we do that, four partitions will appear:

- boot
- rootfs
- usrdata
- data

In rootfs partition, we will find the init.rc file. Now we can edit it with vim or nano.

*Steps*

1) Open init.rc file with nano

2) Search for the lines:

```
service debuggerd /system/bin/debuggerd
    class main
```

Bellow these lines, we are going to write our script initialization:

```
service scriptgpios /system/bin/sh /system/etc/scriptgpios.sh
    class main
    oneshot
```

3) Copy "scriptgpios.sh" from your computer to "/system/etc" in the rootfs partition of the SD card. This file must have execution permissions.

Then we unmount all partitions and put SD card again in the board, when Android boots, this time GPIOs will be configured.

**Tutorial 05:** *Compiling Texas Instruments Android kernel from source.*

If we want to compile Android kernel ( useful if we want to create our custom drivers ) we need to download TI-DevKit from Texas web page:

http://downloads.ti.com/sitara_android/esd/TI_Android_DevKit/TI_Android_JB_4_2_2_DevKit_4_1_1/index_FDS.html

In the “TI Android Sources” section, we download “TI Android JB 4.2.2 DevKitV4.1.1 AM335x Sources”

| TI Android JB 4.2.2 AM335x Sources | |
|---|---|
| TI Android JB 4.2.2 DevKitV4.1.1 AM335x Sources | TI customized sources for Android JB 4.2.2, linux kernel 3.2(with 3D Acceleration and WL127x/WL18xx drivers), boot-strap and u-boot |
| TI-Android-JB-4.2.2-DevKit-4.1.1.xml | TI Android Release Manifest File to be used to clone sources from gitorious.org/rowboat |
| TI Android DevKit Software License manifests | TI Android DevKit software license manifests |

*Steps*

1) The file downloaded is a “.bin” we need to change execution permissions and execute it:

```
chmod +x TI_Android_JB_4.2.2_DevKit_4.1.1.bin
./TI_Android_JB_4.2.2_DevKit_4.1.1.bin
```

We suppose this file is in “~/beagleboneblack” directory, if you are using another path, pay attention and change it in all parts where it appears.

2) Once the binary file decompressed, the directory “TI_Android_JB_4.2.2_DevKit_4.1.1” will appear. Now we can compile the Kernel:

```
cd ~/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/kernel

PATH=$HOME/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin:$PATH

make ARCH=arm CROSS_COMPILE=arm-eabi- distclean

make ARCH=arm CROSS_COMPILE=arm-eabi- am335x_evm_android_defconfig

make ARCH=arm CROSS_COMPILE=arm-eabi- uImage
```

This process can take some of minutes depending of your computer.

3) Kernel image will be created in TI_Android_JB_4.2.2_DevKit_4.1.1/kernel/arch/arm/boot. File's name is “uImage”. Put the SDCard in your card reader and copy this file into “boot” partition.

**Tutorial 06:** *I2C modules. Example with LM75 temperature sensor.*

*Requirements:*

- A board with an LM75 I2C temperature sensor.
- Android NDK installed in your computer.

We are going to configure I2C port and we are going to use it for reading an LM75 temperature sensor.

If we navigate Android directory, in /dev we are going to find “i2c-3” device. This device corresponds with the pins 19 and 20 from P9 connector on the board.

| P9 pin # | I2C function |
|---|---|
| 19 | SCL |
| 20 | SDA |

*Steps*

1) We will connect LM75 chip to this port. Power supply can be taken from the board from pins 1 and 4.

2) We are going to write a C program for use I2C device as a character device for reading LM75. For that, we need to create a project directory: “lm75test”. Then we need to create the “jni” directory inside. Inside jni folder, we are going to create a “Main.c” file.

```
mkdir lm75test
cd lm75test
mkdir jni
cd jni
nano Main.c
```

3) Write the C code using open, read, write, and ioctl functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


        int file;
        char *filename = "/dev/i2c-3";
        if ((file = open(filename, O_RDWR)) < 0)
            exit(1);

        //1 0 0 1 A2 A1 A0 R/W
        int addr = 0x48; // Device address is 0x48
        if (ioctl(file, I2C_SLAVE, addr) < 0)
            exit(1);

        unsigned char buf[10] = {0};
        if(write(file, buf,1) != 1)
            exit(1);
        int r = read(file,buf,2);
        unsigned char tempH = buf[0];
        unsigned char tempL = buf[1];
```

```
unsigned short temp;
temp=tempH;
temp=temp<<1;
temp=temp&0xFFFE;
tempL=tempL>>7;
tempL=tempL&0x0001;
temp=temp|tempL;
float t = ((float)temp)/2;
printf("Temperature:%f\n",t);
```

4) To compile this code we will use Android-NDK, for that reason, we need to create two extra files: “Application.mk” and “Android.mk”

Application.mk:

```
APP_OPTIM     := release
APP_PLATFORM        := android-15
```

Android.mk:

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

include $(CLEAR_VARS)

LOCAL_MODULE    := lm75test
LOCAL_SRC_FILES := Main.c

include $(BUILD_EXECUTABLE)
```

5) We can compile our code now, inside jni directory we execute NDK compiler (the path can change if you are using a newer version)

```
~/adt-bundle-linux-x86_64-20130219/android-ndk-r8d/ndk-build
```

6) If compilation was OK, it will generate the executable file in: lm75test/lib/armeabi/lm75test

7) Copy the executable into Beaglebone's Android etc directory:

```
./adb push lm75test/lib/armeabi/lm75test /etc/
```

8) Enter to Android console, change execution permissions for our program and execute it.

```
./adb shell
cd etc
chmod 777 lm75test
./lm75test
```

The program will print the actual temperature read from LM75.

*Considerations*

If we want, any Android application can access this device, we need to add to the boot script we made before, the permissions for the device:

```
chmod 777 /dev/i2c-3
```

**Tutorial 07:** *UART1 and UART2 modules.*

*Requirements:*

- busybox must be installed

UART 1 and 2 pinout:

| **Connector P9** | |
|---|---|
| PIN 24 | UART1_TXD |
| PIN 26 | UART1_RXD |
| PIN 21 | UART2_TXD |
| PIN 22 | UART2_RXD |

UART1 is managed by ttyO1 device (/dev/ttyO1 )
UART2 is managed by ttyO1 device (/dev/ttyO2)

By default, pins are in MODE7 (GPIO) if we want to use this pins as UARTs we need to change pins mux configuration.

*Steps*

1) We can see UART1 pins mux configuration with the following command:

```
cat /sys/kernel/debug/omap_mux/uart1_rxd
```

2) By default, pin mux is GPIO, we need to change it to UART1 RX

```
echo 20 > /sys/kernel/debug/omap_mux/uart1_rxd
```

Now when we check configuration, it will be:

```
name: uart1_rxd.uart1_rxd (0x44e10980/0x980 = 0x0020), b NA, t NA
mode: OMAP_MUX_MODE0 | AM33XX_PIN_INPUT_PULLDOWN
signals: uart1_rxd | mmc1_sdwp | NA | i2c1_sda | NA | pr1_uart0_rxd_mux1 | NA |
gpio0_14
```

3) Same way we change tx pin configuration.

```
echo 0 > /sys/kernel/debug/omap_mux/uart1_txd
```

4) Now we do the same for UART2, in this case pins are called spi0_d0 and spi0_sclk in Android file system.

```
echo 1 > /sys/kernel/debug/omap_mux/spi0_d0
echo 21 > /sys/kernel/debug/omap_mux/spi0_sclk
```

5) To test UARTs we are going to use “microcom” an application included in busybox similar to “minicom”.

```
/data/busybox/busybox microcom –s 9600 ttyO1

/data/busybox/busybox microcom –s 9600 ttyO2
```

**NOTE:** See steps 14 and 15 from Tutorial 09 for busybox installation.

**Tutorial 08:** *SPI module.*

*Requirements:*

- TI-DevKit must be installed.
- Android Kernel must be compiled.

SPI0 pinout:

| **Connector P9** | |
|---|---|
| PIN 17 | SPI0_CS0 |
| PIN 18 | SPI0_D1 |
| PIN 21 | SPI0_D0 |
| PIN 22 | SPI0_SCLK |

SPI is not enabled by default, we need to edit “board-am335xevm.c” file, enter in kernel 's menuconfig options and enable SPIDEV and re-compile Android kernel with this features, other way SPI device will not appear in /dev directory.

By default, pins are in MODE7 (GPIO) if we want to use these pins as SPI pins, we need to change pins mux configuration.

*Steps*

1) We are going to enable SPI0 because SPI1 is used by HDMI module and it cannot be used. Edit “board-am335xevm.c” file in kernel's directory “arch/arm/mach-omap2/”

```
nano arch/arm/mach-omap2/board-am335xevm.c
```

2) Change “am335x_spi0_slave_info” struct for the follow one:

```
static struct spi_board_info am335x_spi0_slave_info[] = {
	{
		.modalias = "spidev",
		.max_speed_hz = 48000000, //48 Mbps
		.bus_num = 1,
		.chip_select = 0,
		.mode = SPI_MODE_1,
	},
};
```

3) Change “spi0_init” function for the follow one:

```
static void spi0_init(int evm_id, int profile)
{
	setup_pin_mux(spi0_pin_mux);
	spi_register_board_info(am335x_spi0_slave_info,
	ARRAY_SIZE(am335x_spi0_slave_info));
	return;
}
```

4) Add in the “beaglebone_black_dev_cfg[]” array, this line :

```
{spi0_init,		DEV_ON_BASEBOARD, PROFILE_NONE},
```

5) Before compiling kernel, enter in menuconfig options and enable SPI settings:

```
cd ~/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/kernel

PATH=$HOME/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin:$PATH

make ARCH=arm CROSS_COMPILE=arm-eabi- distclean

make ARCH=arm CROSS_COMPILE=arm-eabi- am335x_evm_android_defconfig

make ARCH=arm CROSS_COMPILE=arm-eabi- menuconfig
```

Make sure “McSPI” and “User mode SPI” are enabled.

```
Device Drivers ->
        SPI Support ->
                [*] McSPI driver for OMAP
                [*] User mode SPI device driver support
```

6) Compile kernel

```
make ARCH=arm CROSS_COMPILE=arm-eabi- uImage
```

7) Copy kernel's image into SD card and boot Beaglebone Black. Now device “spidev1.0” should appear in /dev directory.

```
root@android:/dev # ls spi* -la
crw------- root     root      153,   0 2000-01-01 00:00 spidev1.0
```

8) **WARNING**: In tutorial 07, we change pin mux for UART2, this pins are the same for SPI0 clk and d0. Make sure pins spi0_sclk and spi0_d0 are in MODE0 checking mux files:

```
cat /sys/kernel/debug/omap_mux/spi0_sclk
name: spi0_sclk.spi0_sclk (0x44e10950/0x950 = 0x0020), b NA, t NA
mode: OMAP_MUX_MODE0 | AM33XX_PIN_INPUT_PULLDOWN
signals: spi0_sclk | uart2_rxd | i2c2_sda | NA | NA | NA | NA | gpio0_2

cat /sys/kernel/debug/omap_mux/spi0_d0
name: spi0_d0.spi0_d0 (0x44e10954/0x954 = 0x0020), b NA, t NA
mode: OMAP_MUX_MODE0 | AM33XX_PIN_INPUT_PULLDOWN
signals: spi0_d0 | uart2_txd | i2c2_scl | NA | NA | NA | NA | gpio0_3
```

If pins are not as the output above, we need to change them writing a “20” in these files, also we need to disable pull-up in spi0_d0. NOTE: sclk must be an input because a kernel's bug.

```
echo 20 >> /sys/kernel/debug/omap_mux/spi0_sclk
echo 20 >> /sys/kernel/debug/omap_mux/spi0_d0
```

9) We can check SPI module writing a simple C program and connecting spi0_d0 and spi0_d1 physically with a wire. The program will transmit and receive a few bytes. Create a ndk project:

```
mkdir spiTestProject
cd spiTestProject
mkdir jni
cd jni
nano spitest.c
```

10) Write header files and global variables for spi configuration.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <spidev.h>

static const char *device = "/dev/spidev1.0";
static uint8_t mode = SPI_MODE_1;
static uint8_t bits = 8;
static uint32_t speed = 100000; // 100Khz
```

Then we are going to define an error function:

```
static void pabort(const char *s)
{
  perror(s);
  abort();
}
```

Now we write a transfer function for spi test, this function will receive spidev's file descriptor:

```
static void transfer(int fd)
{
	int ret;
	uint8_t tx[] = {0x55,0xC3,0x3C,0x01,0x80,0xFF,0x00};

	uint8_t rx[sizeof(tx)] = {0};

	struct spi_ioc_transfer tr = {
		.tx_buf = (unsigned long)tx,
		.rx_buf = (unsigned long)rx,
		.len = sizeof(tx),
		.delay_usecs = 50,
		.speed_hz = speed,
		.bits_per_word = bits,
		.cs_change=1,
	};

	ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
	if (ret < 1)
		pabort("can't send spi message");

	printf("rx:");
	for (ret = 0; ret < sizeof(tx); ret++)
	printf("%.2X ", rx[ret]);
	printf("\r\n");
}
```

At the end, we write the main function, where we open spidev device and we configure it using ioctl function before calling “transfer” function:

```
int main(int argc, char *argv[])
{
  int ret = 0;
  int fd, counter;

  fd = open(device, O_RDWR);
  if (fd < 0)
    pabort("can't open device");

  // set mode
  ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
  if (ret == -1)
    pabort("can't set spi mode");
  ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
  if (ret == -1)
    pabort("can't get spi mode");
```

```
  // set bits per word
  ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
  if (ret == -1)
    pabort("can't set bits per word");
  ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
  if (ret == -1)
    pabort("can't get bits per word");

  // set max speed hz
  ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
  if (ret == -1)
    pabort("can't set max speed hz");
  ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
  if (ret == -1)
    pabort("can't get max speed hz");

  // Start Test
  printf("spi mode: %d\n", mode);
  printf("bits per word: %d\n", bits);
  printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

  transfer(fd);

  close(fd);

  return ret;
}
```

11) We also need to create Android.mk and Application.mk:

Android.mk

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

include $(CLEAR_VARS)

LOCAL_LDLIBS  := -llog

LOCAL_MODULE  := testSpi
LOCAL_SRC_FILES          := spitest.c

include $(BUILD_EXECUTABLE)
```

Application.mk

```
APP_OPTIM        := release
APP_PLATFORM     := android-15
```

12) Now we can compile this program using android-ndk, in jni directory, execute:

```
~/adt-bundle-linux-x86_64-20130219/android-ndk-r8d/ndk-build
```

6) If compilation was OK, it will generate the executable file in: spiTestProject/libs/armeabi/

7) Copy the executable into Beaglebone's Android etc directory:

```
./adb push spiTestProject/libs/armeabi/testSpi /etc/
```

8) Enter to Android console, change execution permissions for our program and execute it.

```
./adb shell
cd etc
chmod 777 testSpi
./testSpi
```

**Tutorial 09:** *Compiling a custom Android driver as a char device.*

*Requirements:*

- TI-DevKit must be installed
- Android Kernel must be compiled

*Steps*

1) Create a folder for the Kernel Module project

```
mkdir moduleProject
cd moduleProject
```

2) Create a C file named “customModule.c”:

```
nano customModule.c
```

3) We need to write the kernel module, first of all, we include some headers files and defines:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ernesto Gigliotti <ernestogigliotti@gmail.com>");
MODULE_DESCRIPTION("Android Custom Driver");
```

4) Then, we define some constants and the file operations structure, this structure will contain the functions that will be called by the program user when it calls read,write,open,ioctl and close functions.

```
#define MODULE_NAME "customModule"

unsigned int module_major;
struct file_operations fops;
```

5) Now we can write the “init_module” function, this function will be called when the module is installed.

```
int init_module(void) {
	int result;
	// Asign file operation functions
	fops.open = open_module;
	fops.release = release_module;
	//fops.read = read_module;
	//fops.write = write_module;
	//fops.unlocked_ioctl = ioctl_module;
	//_____________________________

	// Dinamic reservation for major number
	result=register_chrdev(0, MODULE_NAME, &fops);

	if (result < 0) {
		printk(KERN_WARNING "androidCustomModule> (init_module) error obtaining major number\n");
		return result;
	}
	module_major = result;

	printk( KERN_INFO "androidCustomModule> (int_module) loaded OK\n");
	return 0;
}
```

6) Write “cleanup_module” function. This function will be called when the module is unloaded:

```
void cleanup_module(void)
{
	unregister_chrdev(module_major, MODULE_NAME);
	printk( KERN_INFO "androidCustomModule> (cleanup_module) unloaded OK\n");
}
```

7) Write the “open_module” function. This function will be called when user's program calls “open” function using our device.

```
int open_module(struct inode *pinode, struct file *pfile)
{
	int minor= MINOR(pinode->i_rdev);
	printk(KERN_INFO "androidCustomModule> (open) minor=%d OK\n",minor);
	return 0;
}
```

8) Write “release_module” function. This function will be called when user's program calls “close” function using our device.

```
int release_module(struct inode *pinode, struct file *pfile)
{
	int minor= MINOR(pinode->i_rdev);
	printk(KERN_INFO "androidCustomModule> (release) minor=%d OK\n",minor);
	return 0;
}
```

9) Before we write the read,write and ioctl functions, we are going to compile this kernel module just with the functions mentioned above. For that, we need to create a Makefile. Create a file named “Makefile” with this content:

Makefile:

```
obj-m +=customModule.o

KERNELDIR ?= ~/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/kernel
PWD := $(shell pwd)
CROSS_COMPILE=~/beagleboneblack/TI_Android_JB_4.2.2_DevKit_4.1.1/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin/
arm-eabi-
ARCH=arm

default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) ARCH=arm CROSS_COMPILE=$(CROSS_COMPILE) modules

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

In this Makefile we pass the kernel's path inside TI-DevKit. Kernel must be compiled. Also we pass the compiler's path. Change these paths if you have TI-DevKit in another place.

10) Write function prototypes at the beginning of the file:

```
int open_module(struct inode *pinode, struct file *pfile);
int release_module(struct inode *pinode, struct file *pfile);
```

11) Now we can compile the module executing “make” command:

```
make
```

File “customModule.ko” will be generated.

12) Now we can install our module into the Android Operating System. Copy the module by adb and install it:

```
./adb push /home/USER/beagleboneblack/moduleProject/customModule.ko /etc
./adb shell
cd /etc
insmod customModule.ko
```

13) The module will be installed and we can see kernel's log messages in kmsg file:

```
cat /proc/kmsg
```

Output will be:

```
<6>[  700.884185] androidCustomModule> (int_module) loaded OK
```

We can unload the module with “rmmod” command

14) Now our module is working, we need to create one or more devices in /dev directory. We need to use “mknod” command for that, but this command is not part of TI Android distribution. We need to install busybox. This application will allow us to use a big set of standard Linux's commands on Android, included “mknod”

Downlod busybox from:
https://gforge.ti.com/gf/project/omapandroid/wiki/?pagename=Installing+Busybox+command+line+tools

15) We need to copy busybox executable file into /data/busybox in Android filesystem:

```
./adb shell
mkdir /data
cd /data
mkdir busybox
CTRL+D ← (quit from Android console)

./adb push /home/USER/beagleboneblack/busybox /data/busybox/
```

16) Now we need to create a script for module's installation ( using insmod and mknod commands ) this script will install the module and it will create the devices in /dev directory:

installModule.sh

```
#!/system/bin/sh
module=customModule
device=/dev/android-custom-dev-
perm=666

rmmod ${module}

insmod ${module}.ko
major=`cat /proc/devices | grep ${module} | /data/busybox/busybox cut -d' ' -f1`

rm -f ${device}*
/data/busybox/busybox mknod ${device}0 c ${major} 0
/data/busybox/busybox mknod ${device}1 c ${major} 1
chmod ${perm} ${device}?
```

In this script we delete the module, then we install it and we get the major number generated. Then the script deletes current devices, and creates the new ones with the major number obtained before.

17) Copy customModule.ko and installModule.sh into /etc on Android by adb:

```
./adb push /home/USER/beagleboneblack/moduleProject/customModule.ko /etc
./adb push /home/USER/beagleboneblack/moduleProject/installModule.sh /etc
```

18) Enter to Android console and execute the script:

```
./adb shell
cd /etc
chmod 777 installModule.sh
./installModule
```

If installation is OK, we will be able to see two new devices in /dev

- /dev/android-custom-dev-0
- /dev/android-custom-dev-1

19) Now the module is working, we are going to complete the module C program with write,read and ioctl functions. Open again customModule.c and add read function:

First the prototype at the begining:

```
static ssize_t read_module(struct file *filp,char *buffer,size_t length,loff_t *offset);
```

Then the function:

```
static ssize_t read_module(struct file *filp,
         char *buffer,    /* The buffer to fill with data */
         size_t length,   /* The length of the buffer     */
         loff_t *offset)  /* Our offset in the file       */
      {
      int bytes_read = 0;

      int minor= MINOR(filp->f_dentry->d_inode->i_rdev);
      printk(KERN_INFO "androidCustomModule> (read) minor: %i \n",minor);

      char message[64]="Hello from module\0";
      char * msg_Ptr = message;
      while (length>0){
            /* put_user copies data from
             * the kernel data segment to the user data segment. */
           put_user(*(msg_Ptr), buffer++);

           length--;
           bytes_read++;
           if(*(msg_Ptr)==0)
             break;
           msg_Ptr++;
      }
      return bytes_read;
}
```

In this module, we are simulating a device which we can read the message “Hello from device”

“put_user” function is used to copy data from kernel space to user space.

Do not forget uncomment fops assignment in init_module function.

20) Write the “write” function and its prototype:

Prototype:

```
static ssize_t write_module(struct file *pfile, const char *buf, size_t size_buf, loff_t *f_pos);
```

Function:

```
static ssize_t write_module(struct file *pfile, const char *buf, size_t size_buf, loff_t *f_pos) {
	unsigned long not_copied;
	int menor= MINOR(pfile->f_dentry->d_inode->i_rdev);

	printk(KERN_INFO "androidCustomModule> (write) minor: %i \n",menor);

	char msgIn[64];
	not_copied=__copy_from_user(msgIn,buf,size_buf); // copy from user space to kernel space
	if (not_copied>0) {
		printk(KERN_WARNING "androidCustomModule> (write) error copying data from user\n");
		return(-EFAULT);
	}
	printk(KERN_INFO "androidCustomModule> (write) msg:%s \n",msgIn);

	(*f_pos)+=size_buf;
	return(size_buf);
}
```

In this case, this function prints by kernel log the message that it receives for writing.

21) The last function we are going to write is “ioctl” this function will allow us to configure our device through commands. We need to create our header file called “customModule.h”

customModule.h:

```
#define CUSTOM_MODULE_IOC_NMAGIC 'c'
#define CUSTOM_MODULE_IOC_CMD_0 _IO(CUSTOM_MODULE_IOC_NMAGIC, 1)
#define CUSTOM_MODULE_IOC_CMD_1 _IO(CUSTOM_MODULE_IOC_NMAGIC, 2)
#define CUSTOM_MODULE_IOC_CMD_2 _IO(CUSTOM_MODULE_IOC_NMAGIC, 3)
#define CUSTOM_MODULE_IOC_CMD_3 _IO(CUSTOM_MODULE_IOC_NMAGIC, 4)
#define CUSTOM_MODULE_IOC_CMD_4 _IOW(CUSTOM_MODULE_IOC_NMAGIC, 5, 1)
```

In our header file we define a “magic number” for the driver and commands for device configuration. User will call ioctl passing this commands to the driver.

Ioctl function:

```
long ioctl_module(struct file *pfile, unsigned int cmd, unsigned long args) {
	int minor= MINOR(pfile->f_dentry->d_inode->i_rdev);
	printk(KERN_INFO "androidCustomModule> (ioctl) minor: %i \n",minor);

	if (_IOC_TYPE(cmd) != CUSTOM_MODULE_IOC_NMAGIC)
		return -EINVAL;

	if (_IOC_NR(cmd) >= 5) // we have only 5 commands
		return -EINVAL;

	printk(KERN_INFO "androidCustomModule> (ioctl) CMD: %d \n", _IOC_NR(cmd));

	switch(cmd) {
		case CUSTOM_MODULE_IOC_CMD_0:
			// ...
			break;
		// Other commands
		//...
		default:
			return -EINVAL;
	}
	return 0;
}
```

22) Compile the module again and copy it into BeagleBone's Android. Now we are going to create a Test program for our device. Create a “deviceTestProject” directory, a jni folder inside, copy the module header file inside jni folder and create test.c file.

```
mkdir deviceTestProject
cd deviceTestProject
mkdir jni
cd jni
cp ~/beagleboneblack/moduleProject/customModule.h .
nano test.c
```

23) Write a test program using open, read, write and ioctl functions on our device:

test.c:

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "customModule.h"

int main() {
	int file0, result;
	printf("Driver Test\n");

	file0=open("/dev/android-custom-dev-0",O_RDWR);
	if (file0==-1) return(-1);

	char msg[64];
	int r = read(file0,msg,64);
	printf("Read msg: %s \n",msg);

	char buffer[64]="Message from user\0";
	write(file0,buffer,64);

	result=ioctl(file0, CUSTOM_MODULE_IOC_CMD_0,NULL);
	if (result==0)
		printf("ioctl OK\n");
	else
		printf("ioctl ERROR\n");

	close(file0);
	return 0;
```

24) Compile this test program using NDK. Do not forget create Application.mk and Android.mk files first:

Application.mk:

```
APP_OPTIM	:= release
APP_PLATFORM	:= android-15
```

Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE	:= deviceTest
LOCAL_SRC_FILES := test.c
include $(BUILD_EXECUTABLE)
```

Go to jni folder and compile the program using NDK:

```
~/adt-bundle-linux-x86_64-20130219/android-ndk-r8d/ndk-build
```

25) Copy executable file in /libs/armeabi into BeagleBoard's Android by adb:

```
./adb push
/home/USER/beagleboneblack/deviceTestProject/libs/armeabi/deviceTest /etc/
```

26) Go to Android console and execute deviceTest program:

```
./ adb shell
cd etc
chmod 777 deviceTest
./deviceTest
```

Program's Output will be:

```
Driver Test
Read msg: Hello from module
ioctl OK
```

Kernel log's Output will be:

```
<6>[ 5295.389801] androidCustomModule> (int_module) loaded OK
<6>[ 5309.067596] androidCustomModule> (open) minor=0 OK
<6>[ 5309.074859] androidCustomModule> (read) minor: 0
<6>[ 5309.080596] androidCustomModule> (write) minor: 0
<6>[ 5309.085845] androidCustomModule> (write) msg:Message from user
<6>[ 5309.092895] androidCustomModule> (ioctl) minor: 0
<6>[ 5309.098052] androidCustomModule> (ioctl) CMD: 1
<6>[ 5309.103607] androidCustomModule> (release) minor=0 OK
```

*Considerations*

If we need out custom devices always available, we need to call “installModule.sh” script when Android is booting, this is possible adding it to the boot script we already made.

**Tutorial 10:** *Accessing to hardware from custom Android driver.*

*Requirements:*

- Custom driver must already work.

Until now, our driver just can manage a local ram memory portion without interaction with hardware, if we want to read/write processor's registers for controlling hardware modules we can not do this directly, we need to use functions to convert physical addresses to virtual addresses.

*Steps*

1) Use “request_mem_region” function to reserve physical addresses for accessing. This reservation should be done in module's init function.

```
struct resource *res = request_mem_region(PHYS_ADDR,4,"area_name");
if(res!=NULL)
{
   // Reservation OK
}
```

In this example, we are reserving 4 bytes starting from address “PHYS_ADDR” and we are assigning to this area the name “area_name”

When memory area is reserved, we can check that using:

```
cat /proc/iomem
```

2) Same way, we should release this memory area in module's close function:

```
release_mem_region(PHYS_ADDR,4);
```

3) Now we can obtain a pointer for read/write using “ioremap_nocache” function

```
unsigned int * addr = (unsigned int *)ioremap_nocache(PHYS_ADDR,4);
```

4) Using ioread32 and iowrite32 we can read/write this memory address:

```
unsigned int value = ioread32(addr);
value=value | (1<<21);
iowrite32(value,addr);
```

5) Release the pointer:

```
iounmap(addr);
```

This way we can read/write physical addresses in our driver and use processor's hardware.

This headers files must be included:

```
#include <linux/ioport.h>
#include <asm/io.h>
```

**Tutorial 11:** *Compiling kernel 3.8 with device tree support for Android.*

Kernel 3.2 does not have Device Tree support, we cannot use PWM module or ADC if we do not initialize these modules using Device Tree, because of this, we are going to left TI kernel and filesystem aside, and we are going to use rowboat project to compile Android and Robert Nelson's kernel 3.8.

*Steps*

1) Fetch rowboat's Android project for BBB:

```
mkdir RobertNelsonKernel
cd RobertNelsonKernel
repo init -u git://gitorious.org/rowboat/manifest.git -m rowboat-jb-am335x.xml
repo sync
```

2) Fetch and compile Robert Nelson Kernel

```
git clone https://github.com/RobertCNelson/bb-kernel.git
cd linux-dev
git checkout origin/am33x-v3.8 -b tmp
./build_kernel.sh
```

Kernel compilation will start and we will have to add some features in kernel's menuconfig:

```
Device Drivers ->
        [*] Staging drivers ->
                Android ->
                        <*> Android log driver
```

3) Now we are going to fetch Android's bootloader “U-Boot”. In RobertNelsonKernel directory:

```
git clone git://git.denx.de/u-boot.git
cd u-boot/
git checkout v2013.04 -b tmp

wget
https://raw.github.com/eewiki/u-boot-patches/master/v2013.04/0001-am335x_evm-uEn
v.txt-bootz-n-fixes.patch

patch -p1 < 0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
```

4) We compile U-boot with the following commands:

```
make ARCH=arm
CROSS_COMPILE=~/RobertNelsonKernel/linux-dev/dl/gcc-linaro-arm-linux-gnueabihf-4
.7-2013.04-20130415_linux/bin/arm-linux-gnueabihf- distclean

make ARCH=arm
CROSS_COMPILE=~/RobertNelsonKernel/linux-dev/dl/gcc-linaro-arm-linux-gnueabihf-4
.7-2013.04-20130415_linux/bin/arm-linux-gnueabihf- am335x_evm_config

make ARCH=arm
CROSS_COMPILE=~/RobertNelsonKernel/linux-dev/dl/gcc-linaro-arm-linux-gnueabihf-4
.7-2013.04-20130415_linux/bin/arm-linux-gnueabihf-
```

5) Now we need to compile Android Filesystem using our new Kernel 3.8, for that reason we need to change kernel link in rowboat project:

```
cd rowboat-android
mv kernel kernel.backup
ln -s ~/RobertNelsonKernel/linux-dev/KERNEL kernel
```

And we need to do some modifications in makefile:

```
chmod 644 Makefile
```

We need to locate the following line:

```
export PATH :=$(PATH):$(ANDROID_INSTALL_DIR)/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin
```

Comment out that line and replace it with the following:

```
export
PATH :
=~/RobertNelsonKernel/linux-dev/dl/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-2
0130415_linux/bin:$(PATH)

export CC_PREFIX := arm-linux-gnueabihf-
```

Now locate this line:

```
$(MAKE) -C hardware/ti/wlan/mac80211/compat_wl18xx ANDROID_ROOT_DIR=$
(ANDROID_INSTALL_DIR) CROSS_COMPILE=arm-eabi- ARCH=arm install
```

And we need to replace it for:

```
$(MAKE) -C hardware/ti/wlan/mac80211/compat_wl18xx ANDROID_ROOT_DIR=$
(ANDROID_INSTALL_DIR) CROSS_COMPILE=$(CC_PREFIX) ARCH=arm install
```

Now we can compile Android's filesystem, you can add '-j2' or '-j4' if you have many cores in your computer, this will acelerate the process.

```
make TARGET_PRODUCT=beagleboneblack OMAPES=4.x droid
```

6) In this step we are going to create the root filesystem. In rowboat-android directory:

```
cd out/target/product/beagleboneblack
mkdir android_rootfs
cp -r root/* android_rootfs
cp -r system android_rootfs
cd android_rootfs/system
tar -xvzf ~/RobertNelsonKernel/linux-dev/deploy/*modules.tar.gz
```

Assuming that we are still in the out/target/product/beagleboneblack directory, we need to backup the file android_rootfs/**init.rc** and then open it in a text editor, we are going to to change init.rc file:

In the line:

```
import /init.${ro.hardware}.rc
```

we are going to change it by:

```
import /init.am335xevm.rc
```

Then, we need to change the file android_rootfs/**fstab.am335xevm**, the first line is:

```
/dev/block/platform/omap/omap_hsmmc.0/mmcblk0p3
```

and we will change it by:

```
/dev/block/mmcblk0p3
```

Now we need to disable graphical aceleration in file android_rootfs/system/**build.prop** because this is not available in 3.8 kernel.

```
debug.egl.hw=0
video.accelerate.hw=0
```

Finally, in the out/target/product/beagleboneblack directory, we execute:

```
../../../../build/tools/mktarball.sh ../../../host/linux-x86/bin/fs_get_stats
android_rootfs . rootfs rootfs.tar.bz2
```

The rootfs.tar.bz2 file will be created. This is the Android's filesystem we are going to use.

7) Now we are going to create an “image” directory and gather all necessary files there:

```
cd ~/RobertNelsonKernel
mkdir image
cp rowboat-android/external/ti_android_utilities/am335x/mk-mmc/* image
cp rowboat-android/out/target/product/beagleboneblack/rootfs.tar.bz2 image
cp u-boot/MLO image
cp u-boot/u-boot.img image
cp linux-dev/KERNEL/arch/arm/boot/zImage image
cp linux-dev/KERNEL/arch/arm/boot/dts/am335x-boneblack.dtb image
```

8) We need to create u-boot configuration file uEnv.txt in image directory with the following content:

```
kernel_file=zImage
console=ttyO0,115200n8
mmcroot=/dev/mmcblk0p2 rw
mmcrootfstype=ext4 rootwait
loadkernel=load mmc ${mmcdev}:${mmcpart} 0x80200000 ${kernel_file}
loadfdt=load mmc ${mmcdev}:${mmcpart} 0x815f0000 ${fdtfile}
boot_ftd=run loadkernel; run loadfdt
mmcargs=setenv bootargs consoleblank=0 console=${console}
androidboot.console=ttyO0 mem=512M root=${mmcroot} rootfstype=${mmcrootfstype}
init=/init ip=off video=720x480-16@60 qemu=1 vt.global_cursor_default=0
uenvcmd=run boot_ftd; run mmcargs; bootz 0x80200000 - 0x815f0000
```

9) We can create out uSD card now. First we need to know what sdX our sd card is. Then we are going to run the script mkmmc-android.sh

```
cd image

./mkmmc-android.sh [YOUR DEVICE FILE FOR THE MICROSD CARD] MLO u-boot.img zImage
uEnv.txt rootfs.tar.bz2
```

10) Now we mount “boot” partition of our sd card and we do these changes:

```
cd /media/boot
sudo mv uImage zImage
sudo cp ~/RobertNelsonKernel/image/am335x-boneblack.dtb .
sync
```

We added device tree file and we changed uImage for zImage.

Now our uSD card is ready to run Android with kernel 3.8 and device tree support.

**NOTE:** adb does not work by USB, we need to connect Beaglebone Black using Ethernet connector.

**Tutorial 12:** *Using PWM modules.*

*Requirements:*

- Robert Nelson 3.8 Kernel compiled.
- Android running with Robert Nelson 3.8 Kernel and Device Tree

We need Android running with kernel 3.8 for using PWM modules because of we are going to enable PWM using device tree.

We have 8 PWM outs:

| PIN | PWM number |
|---|---|
| P9 - 22 | PWM0 |
| P9 - 21 | PWM1 |
| P9 - 42 | PWM2 |
| P9 - 14 | PWM3 |
| P9 - 16 | PWM4 |
| P8 - 19 | PWM5 |
| P8 - 13 | PWM6 |
| P9 - 28 | PWM7 |

*Steps*

1) First of all we need to install the pwm kernel module using “insmod” command. This module is in /system/lib/modules/3.8.13-bone40.1/kernel/drivers/pwm/ and it is called “pwm_test.ko”.

```
insmod /system/lib/modules/3.8.13-bone40.1/kernel/drivers/pwm/pwm_test.ko
```

2) Using device tree now we can enable PWM support:

```
echo am33xx_pwm > /sys/devices/bone_capemgr.9/slots
```

3) PWM module is ready to use. Now we can export PWM outs as we did with GPIOs. If we want to enable PWM6 we need to write:

```
echo 6 > /sys/class/pwm/export
echo bone_pwm_P8_13 > /sys/devices/bone_capemgr.*/slots
```

Where “6” is for PWM6 and “bone_pwm_P8_13” is because of PWM6 corresponds with P8_13.

4) Now “pwm6” directory will appear, we will find there four important files:

- **period_ns**: Here we set pwm's period in nanoseconds.
- **duty_ns**: Here we set pwm's duty in nanoseconds.
- **polarity**: Here we set pwm's polarity..
- **run**: When we write “1” PWM starts running, and when we write “0” PWM stops.

5) Finally, we change files's permissions to 666, in this way we can write these files using a C program or by bash console.

```
chmod 666 pwm/pwm6/period_ns
chmod 666 pwm/pwm6/duty_ns
chmod 666 pwm/pwm6/polarity
chmod 666 pwm/pwm6/run
```

The follow bash script installs pwm module and enables all PWM outs, we can run it on Android's initialization.

```
#!/system/bin/sh

#Identify bone version
module=$(echo /system/lib/modules/3.8.13-bone*/kernel/drivers/pwm/pwm_test.ko | /data/busybox/busybox awk
'{print $1}')

#load pwm module
insmod $module

# Identify cape-manager slots
slot=$(echo /sys/devices/bone_capemgr.*/slots | /data/busybox/busybox awk '{print $1}')
pwm=/sys/class/pwm/

# Load PWM module
if [ $(cat $slot | grep am33xx_pwm -c) -gt 0 ]; then
   echo am33xx_pwm already loaded, not loading again;
else
   echo Enabling PWM driver am33xx_pwm
   echo am33xx_pwm > $slot
fi;

# Safe pins that won't interfere with one another
#ports=(P9_22 P9_42 P9_16 P8_13 P9_28);
#portnumbers=(0 2 4 6 7);
#Full correspondence from pwm0-pwm7
ports=(P9_22 P9_21 P9_42 P9_14 P9_16 P8_19 P8_13 P9_28);
portnumbers=(0 1 2 3 4 5 6 7);
# Enable PWM pins
# They must be exported at this stage, before the device tree
# overlay for that pin is enabled.
c=0
while [ c -lt ${#ports[*]} ]
do
   if [ ! -d $pwm/pwm${portnumbers[$c]} ]; then
           echo Exporting PWM ${portnumbers[$c]} \(${ports[$c]}\)
      echo ${portnumbers[$c]} > $pwm/export
           chmod 666 $pwm/pwm${portnumbers[$c]}/period_ns
           chmod 666 $pwm/pwm${portnumbers[$c]}/duty_ns
           chmod 666 $pwm/pwm${portnumbers[$c]}/polarity
           chmod 666 $pwm/pwm${portnumbers[$c]}/run
   else
           echo PWM ${portnumbers[$c]} already enabled
   fi;

   if [ $(cat $slot | grep ${ports[$c]} -c) -gt 0 ]; then
           echo PWM pin ${ports[$c]} already enabled, not enabling again
   else
      (echo bone_pwm_${ports[$c]} > $slot) 1>&2 >> /dev/null
   fi;
   c=$((c+1))
done
```

We can change which PWM modules we want to initialize

**Tutorial 13:** *Using ADC module.*

*Requirements:*

- Robert Nelson 3.8 Kernel compiled.
- Android running with Robert Nelson 3.8 Kernel and Device Tree

Beaglebone black has 7 analog inputs, we need to enable them if we want to use them. Analog inputs driver is similar to GPIO driver, we will read a file where the analog value is. We will have a file for each input, the value is a number between 0 and 4095 (12 bit resolution).

We have 7 Analog inputs:

| PIN | Analog number |
|---|---|
| P9 - 39 | AIN0 |
| P9 - 40 | AIN1 |
| P9 - 37 | AIN2 |
| P9 - 38 | AIN3 |
| P9 - 33 | AIN4 |
| P9 - 36 | AIN5 |
| P9 - 35 | AIN6 |

*Steps*

1) Enable analog inputs using device tree:

```
echo BB-ADC > /sys/devices/bone_capemgr.*/slots
```

2) Now we will find 7 files in "/sys/bus/platform/drivers/bone-iio-helper/helper.15" AIN0 to AIN6. If we want to read Analog 0 input we can use "cat" command:

```
cd /sys/bus/platform/drivers/bone-iio-helper/helper.15
cat AIN0
```

## Bibliography


- Texas Instruments Android images
  - http://downloads.ti.com/sitara_android/esd/TI_Android_DevKit/TI_Android_JB_4_2_2_DevKit_4_1_1/index_FDS.html
- Texas Instruments Android DevKit Guide
  - http://processors.wiki.ti.com/index.php/AM335X_EVM-SK_Android_Devkit_Guide
- Enable SPIDEV:
  - http://communistcode.co.uk/blog/blogPost.php?blogPostID=1
- Configure UARTS:
  - http://www.gigamegablog.com/2012/01/22/beaglebone-coding-101-using-the-serial-and-analog-pins/
- Create kernel module
  - http://www.it.uc3m.es/alcortes/docs/apuntes_lao/lao/node68.html
- “BeagleBone Black System . Reference Manual” - Gerald Coley, October 2013.
- “AM335x ARM® CortexTM-A8 Microprocessors. Technical Reference Manual” - Texas Instruments, October 2011.
- Compile kernel 3.8 for Android
  - http://icculus.org/~hendersa/android/
- PWM bonecape manager usage
  - https://github.com/szmoore/MCTX3420/wiki/Software%3A-PWM-Control