Processor SDK Linux Training: Introduction to Device Driver Development

Contents

Introduction

NOTE: Commands to be executed for each step of this guide will be marked in **BOLD.**

This lab and accompanying lecture was adapted from Free Electron's Embedded Linux kernel and driver development training (http://free-electrons.com/training/kernel/).

Lab Configuration

The following are the hardware and software configurations for this lab. The steps in this lab are written against this configuration. The concepts of the lab will apply to other configurations but will need to be adapted accordingly.

Hardware

- BeagleBone Black Board - Order Now (http://beagleboard.org/boards)
- Nintendo Nunchuk with UEXT Connector - Order Now (https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/open-source-hardware)
- miniUSB to USB Cable
- USB Serial Cable (female ends)
- Jumper Cables
- microSD Card
- microSD to SD Card Adapter and SD Card Reader **or** a microSD Card Reader
- 2 Ethernet cables
- Router

Lab 1: Setup

Description

This lab will instruct on how to install, setup, and navigate the SDK, the board and its communication, and the U-Boot environment. These steps are required for the remainder of the labs.

Lab Steps

Processor SDK Setup

Download and Install the SDK

1. First, run the command below to ensure that your sources are up to date:

   sudo apt-get update

2. You will also need to install git for this lab:

   sudo apt-get install git

3. Install the Processor Linux SDK package with these instructions.

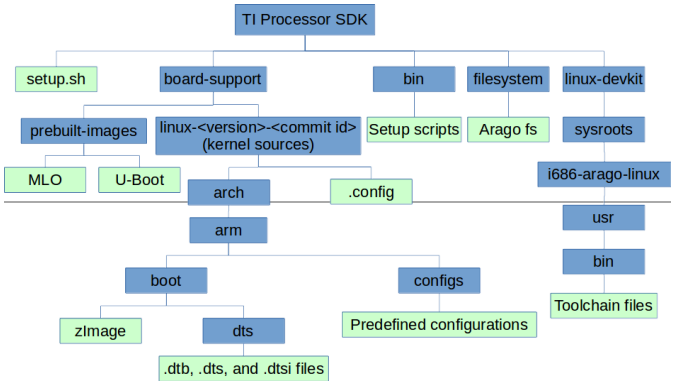
   > **NOTE**
   >
   > There is an additional link on the Processor Linux SDK installer page with steps to complete if you are running a 64 bit version of Linux.

Explore the SDK

Navigate to the newly installed SDK directory in your home directory. Again, the actual filename will be based on the SDK version you download but could be for example ti-processor-sdk-linux-am335x-evm-02.00.00.00. You are encouraged to explore the SDK and Linux kernel sources at this point.

The accompanying chart is a visual representation of the file paths of the important files and directories discussed in the lecture. It is by no means a representation of the entire SDK. It is simply a tool to help you locate resources.

> **NOTE**
>
> For later in the lab, observe that the root of you Linux kernel sources refers to the /board-support/linux-<version>-<commit id> directory of the SDK.



Setup the SDK

1. Starting with Ubuntu 12.04, the user needs to be a part of the dialout group to access the serial port which we will use to communicate with the BeagleBone. Therefore, you need to add your user to the dialout group:

   sudo adduser $USER dialout

2. Log off and log back on in order for this change to take effect.
3. Navigate to the /bin folder in your SDK directory and run **./setup-package-install.sh** to install all additional packages you will need during this lab.

Unpack the File System

1. Create a folder called trainingNFS in the SDK directory.
2. Navigate to /filesystem in your SDK directory and unpack the Arago file system into your trainingNFS directory.

   sudo tar -xzvf arago-base-tisdk-image-am335x-evm.tar.gz -C <SDK path>/trainingNFS

   > **NOTE**
   >
   > <SDK path> refers to the SDK version directory, for example /home/sitara/ti-processor-sdk-linux-am335x-evm-02.00.00.00. This will be used throughout the lab.

3. Navigate to <SDK path>/trainingNFS/boot and delete all files in this directory, as you will rebuild and replace the kernel and its accompanying .btd file in the next step of this lab.

Recompile the Kernel

1. Navigate to the root of your Linux kernel sources, noted earlier in the lab as <SDK path>/board-support/linux-<version>-<commit id>. A git repository should already exist here. Make sure that everything is up to date by running **git status**. Then create and switch to a new branch called training_kernel:

   git checkout -b training_Kernel

2. In order to use the cross compiler toolchain files arm-linux-gnueabihf- you must first define where these files can be found, meaning you will need to export the path in the terminal. You will need to do this each time you open a new terminal window and intend to use the make command:

   export PATH=$PATH:<SDK path>/linux-devkit/sysroots/<Arago Linux>/usr/bin/

   > **NOTE**
   >
   > <Arago Linux> refers to the Arago filepath, for example x86_64-arago-linux. Be sure to include /home/<user>/ before your SDK directory.

   > **NOTE**
   >
   > If you forget to export the path and attempt to use the make command, you will see either an error that says:
   > make: *** No rule to make target "<target>". Stop.
   > or
   > arm-linux-gnueabihf-gcc: command not found.

3. Before you compile your kernel for the first time, you should clean all previously built versions and configurations with the following command:

   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean

   > **NOTE**
   >
   > In the future, unless you are changing the configuration, you should clean the directory with the command **make clean** instead, which will remove previously build versions, but leave your .config file.

   > **IMPORTANT**
   >
   > You will need to define the ARCH and CROSS_COMPILE variables each time you use the make command.

4. Now that you have cleaned your directory and removed your old configuration, you will need to remake your .config file. This lab will use a predefined configuration tisdk_am335x-evm_defconfig, so make the .config file with the following command:

   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- tisdk_am335x-evm_defconfig

   > **NOTE**
   >
   > Older SDK versions may use the configuration singlecore-omap2plus_defconfig

5. Use the following commands to rebuild the kernel and its accompanying files. Run these commands one at a time.

   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- am335x-boneblack.dtb
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules
   sudo make INSTALL_MOD_PATH=<SDK path>/trainingNFS ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules_install

The commands in more detail:

make zImage builds the kernel image binary file zImage. This command will take some time to complete and will generate a long series of outputs.
make am335x-boneblack.dtb builds the accompanying .dtb file necessary for the BeagleBone Black board. More detail on this file will be included in future lectures.
make modules builds the dynamic modules specified by the configuration. More details on dynamic modules will be included in the following lecture.
make modules_install copies the dynamic modules to their proper location in the root file system.

6. Copy over the new zImage (in the arch/arm/boot folder) and the am335x-boneblack.dtb file (in the arch/arm/boot/dts folder) to the <SDK path>/trainingNFS/boot directory.

MicroSD Card Setup

1. Connect your microSD card to your Linux host using the adapter and card reader.
2. Set up your SD card by running the create-sdcard.sh script with sudo (administrative privilege) found in the /bin folder of your SDK directory.

   sudo ./create-sdcard.sh

3. Follow the instructions as prompted, selecting your microSD card (which should be device number 1), choosing yes to partition the card, and selecting 2 partitions. For additional information, see Processor SDK Linux create SD card script. When prompted to either continue installing the filesystem or safely exit the script, **select no** in order to exit the script.
4. You should check to make sure that you now have a microSD card with a boot partition and a rootfs partition, both of which should be empty. The microSD card and its partitions can be accessed under /media/<user>. You may need to eject and reinsert the SD card adapter in order to access the microSD card as the script will have unmounted the card.
5. Navigate to /board-support/prebuilt-images in your SDK directory. Copy the MLO and U-Boot image to the boot partition of your microSD card.

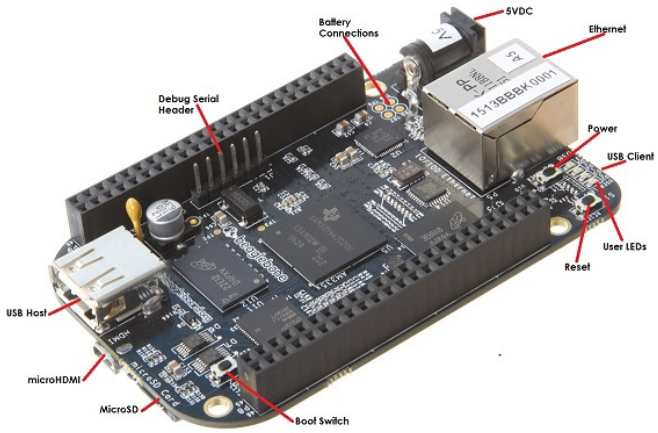
   sudo cp MLO-am335x-evm /media/<user>/boot/MLO
   sudo cp u-boot-am335x-evm.img /media/<user>/boot/u-boot.img

6. Eject the SD card and insert it into the SD card slot on the board.

Board Setup

Getting Familiar with the Board

Take some time to read about the board features and connectors on http://www.elinux.org/Beagleboard:BeagleBoneBlack (http://www.elinux.org/Beagleboard:BeagleBoneBlack). Ensure you know how to properly power off and on the board without damaging it. Do not abruptly cut off power to your BeagleBone.



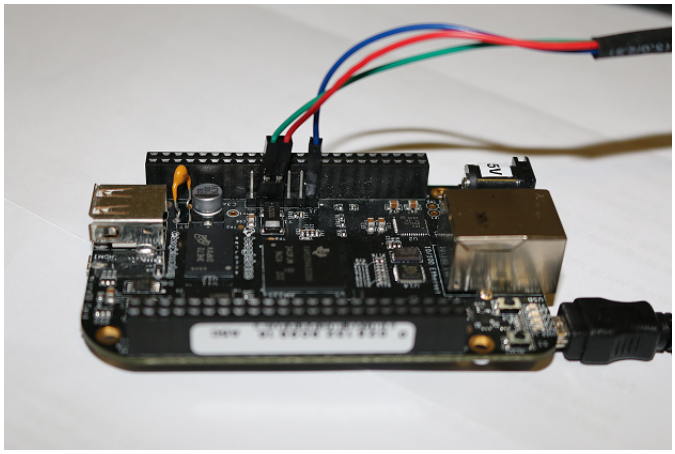
Important things to note are the serial header, the miniUSB port (USB Client), the Ethernet port, the microSD card slot, the reset button, and the power button.

Other Useful Technical Documentation

- Board System Reference Manual (https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true)
- TI AM335x SoC Datasheet (http://www.ti.com/lit/ds/symlink/am3359.pdf)
- TI AM3359 SoC Technical Reference Manual (http://www.ti.com/product/am3359)

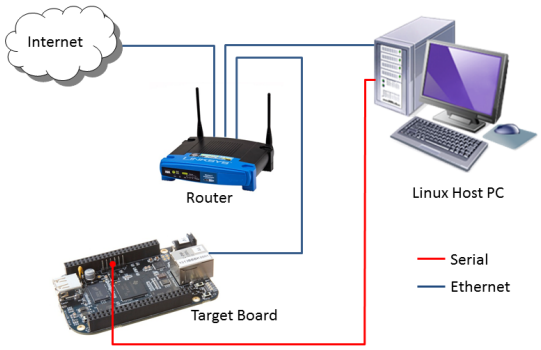
Setup Communication with the Board

1. The BeagleBone serial connector is exported on the 6 pins close to one of the 48 pins headers. Using your USB to Serial adapter (female ends), connect the ground wire (blue) to the pin closest to the power supply connector, and the TX (red) and RX (green) wires to the pins board RX and board TX. The setup is shown below.



> **NOTE**
>
> You should always make sure that you connect the TX pin of the cable to the RX pin of the board, and vice versa.

2. Once the USB to Serial connector is plugged in, a new serial port should appear: /dev/ttyUSB0. You can also see this device appear by looking at the output of dmesg.
3. Ensure that your microSD card is inserted into the microSD card slot on the board. Power up your board by plugging your board into the Linux machine with the miniUSB cable.
4. Finally, use your Ethernet cables to connect the Ethernet port of your board to your router and your router to your Linux machine, as shown in the diagram.



Board Communication Setup

Setup NFS

1. Use **sudo** to edit the file called *exports in* /etc. *Add the following line to the bottom. This will allow NFS to locate the directory you wish to export via NFS, where you wish to export the directory (the * means all IP addresses. You can replace it with the IP address of your board if you wish), and the permissions you wish to use while exporting.*

   <SDK path>/trainingNFS *(rw,nohide,insecure,no_subtree_check,async,no_root_squash)

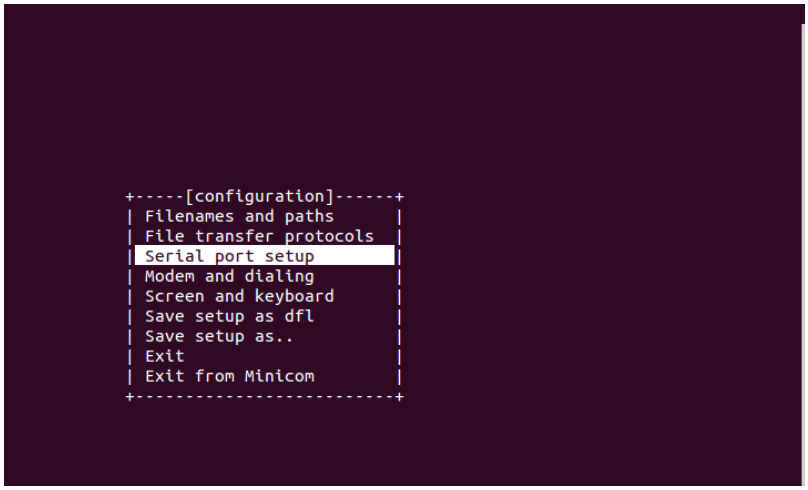
2. Stop and restart the NFS server, which was installed when you ran the setup package script. Either enter the lines one at a time, or add a sleep 1 in between the 2 lines.

   sudo /etc/init.d/nfs-kernel-server stop
   sudo /etc/init.d/nfs-kernel-server start

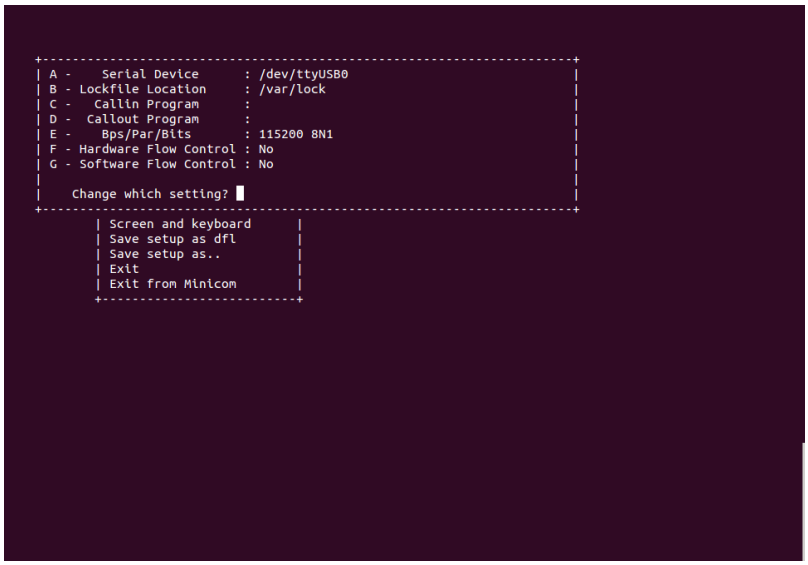
Setup Minicom

1. Minicom is the serial communication program that was installed when you ran the setup-package.sh script earlier in the lab. You will use this program for interfacing with the board over the USB to serial cable. You will need to configure minicom the first time you run it. Start minicom into its configuration menu with the command:

   sudo minicom -s



2. Scroll to and select the Serial port setup option. Press **a** to edit the Serial Device field to /dev/ttyUSB0, then press **Enter**. Press **e** to edit the Bps/Par/Bits field to 115200 8N1, then press **Enter**. Ensure both the Hardware and Software Flow Control fields are set to No. Your settings should now look like the ones in the image below.



3. Press **Enter** to return to the main configuration menu and select Save setup as dfl. A confirmation message should appear.
4. Select Exit from Minicom.

Boot the Board

1. Run the command ifconfig to find the ip address of your host machine. An example is shown below.

```
train@Turtur:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:26:b9:f7:98:5b
          inet addr:128.247.125.30  Bcast:128.247.125.255  Mask:255.255.254.0
          inet6 addr: fe80::226:b9ff:fef7:985b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:52952 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23647 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:44909201 (44.9 MB)  TX bytes:3124023 (3.1 MB)
          Interrupt:20 Memory:f6900000-f6920000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:366 errors:0 dropped:0 overruns:0 frame:0
          TX packets:366 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:31324 (31.3 KB)  TX bytes:31324 (31.3 KB)

wlan0     Link encap:Ethernet  HWaddr c0:cb:38:06:73:ac
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

train@Turtur:~$
```

2. Open the serial communication program Minicom by calling:

   minicom

   > **NOTE**
   >
   > You can exit minicom at any time by hitting the keys Ctrl and A simultaneously, and then hitting X.

3. Reboot your board by pressing the reboot button and when the line Hit any key to stop autoboot shows up, press any button on your keyboard to stop the U-boot countdown. You should now see the U-Boot prompt U-Boot# as shown below. If you miss the prompt, you can always hit the reboot button again.

```
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Jan  1 2014, 17:13:19.
Port /dev/ttyUSB0, 17:04:04

Press CTRL-A Z for help on special keys


U-Boot SPL 2014.07-00107-ga6ef75a (May 22 2015 - 13:27:45)
reading args
spl_load_image_fat_os: error reading image args, err - -1
reading u-boot.img
reading u-boot.img


U-Boot 2014.07-00107-ga6ef75a (May 22 2015 - 13:27:45)

I2C:   ready
DRAM:  512 MiB
NAND:  0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
reading uboot.env
Net:   cpsw, usb_ether
Hit any key to stop autoboot:  0
U-Boot#

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB0
```

4. You can now use the command setenv <target variable> <value> to change your U-Boot environment variables to tell U-Boot to use NFS to mount your kernel and file system before booting. You will need to set the following variables:

   setenv serverip <your host ip>
   setenv rootpath <SDK path>/trainingNFS
   setenv bootfile zImage
   setenv ip_method dhcp
   setenv nfs_bootfile 'nfs ${loadaddr} ${serverip}:${rootpath}${bootdir}/${bootfile}'
   setenv nfs_fdtfile 'nfs ${fdtaddr} ${serverip}:${rootpath}${bootdir}/${fdtfile}'
   setenv bootcmd 'setenv autoload no; dhcp; run nfs_bootfile; run findfdt; run nfs_fdtfile; run netargs; bootz ${loadaddr} - ${fdtaddr}'

   > **NOTE**
   >
   > You can enable line wrapping in minicom by pressing Ctrl-A Z and then W each time the window is opened.

   bootcmd in more detail:

   setenv autoload no - prevents U-Boot from trying to autoload an image over TFTP when you use the dhcp command
   dhcp - discovers the ip address of your board
   run nfs_bootfile - runs the command nfs ${loadaddr} ${serverip}:${rootpath}${bootdir}/${bootfile} stored in the nfs_bootfile variable, which created in an earlier line. This command mounts your zImage file over NFS from your host IP and specified file path to the address in loadaddr
   run findfdt - scans your board for the name of the necessary .dtb file to use
   run nfs_fdtfile - runs the command nfs ${fdtaddr} ${serverip}:${rootpath}${bootdir}/${fdtfile} stored in the nfs_fdtfile variable, which created in an earlier line. This command mounts the necessary .dtb file over NFS from your host IP and specified file path to the address in fdtaddr
   run netargs - sets bootagrs, a boot variable to be automatically passed to the kernel
   bootz ${loadaddr} - ${fdtaddr} - boots the board from the zImage and .dtb file in the loadaddr and fdtaddr memory locations respectively

> **NOTE**
>
> The printenv and help commands are useful for understanding these variables in more depth. If you wish to view the current value of a specific environemtn variable, you can call echo $<target>. editenv <target> is another helpful command to know that can be used to edit an environment variable without the need to completely overwrite it.

5. Save the changes to your uboot environment variables with the command **saveenv**. You should see a message like the one below.

```
U-Boot# saveenv
Saving Environment to FAT...
writing uboot.env
done
U-Boot#
```

If you do not see this message, your board may be using the pre-installed version of U-Boot saved on the internal eMMC rather than the version you copied to your microSD card. You can follow the instructions here to wipe the eMMC so your board will boot from the microSD card.

6. Once your environment variables are saved, reboot your board again.

> **NOTE**
>
> If your ip address changes, you will need to reset the serverip U-Boot variable or your board will not boot.

7. If your board boots correctly, you should see a series of ###, as seen in the image below.

```
Using cpsw device
File transfer via NFS from server 128.247.125.30; our IP address is 128.247.1257
Filename '/home/train/ti-processor-sdk-linux-am335x-evm-01.00.00.00/targetNFS//.
Load address: 0x82000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ############################
done
Bytes transferred = 4460280 (440ef8 hex)
link up on port 0, speed 100, full duplex
Using cpsw device
File transfer via NFS from server 128.247.125.30; our IP address is 128.247.1257
Filename '/home/train/ti-processor-sdk-linux-am335x-evm-01.00.00.00/targetNFS//.
Load address: 0x88000000
Loading: ######
done
Bytes transferred = 30149 (75c5 hex)
Kernel image @ 0x82000000 [ 0x000000 - 0x440ef8 ]
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Device Tree to 8fff5000, end 8ffff5c4 ... OK

Starting kernel ...
```

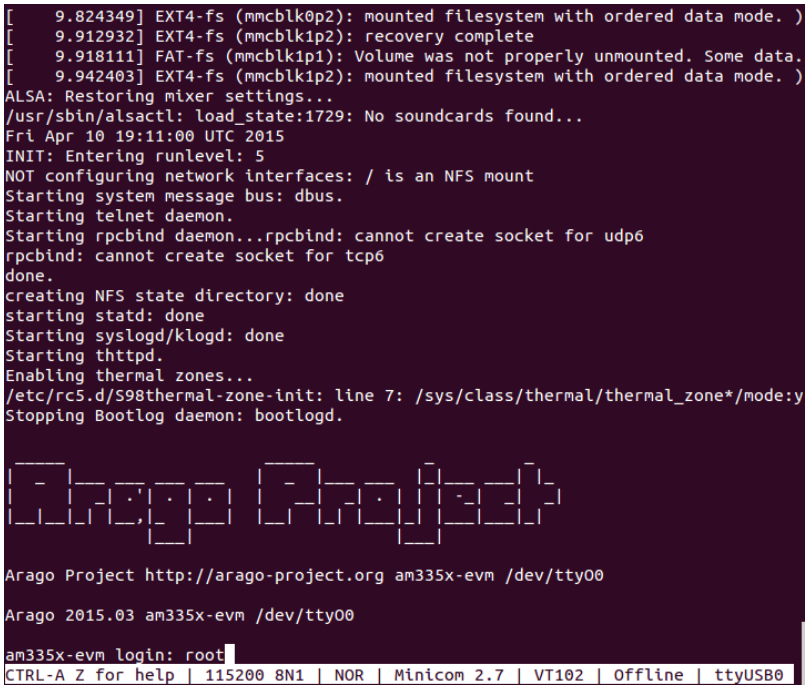
If your board boots incorrectly, you will see a series of TTT, as seen below.

```
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Jan  1 2014, 17:13:19.
Port /dev/ttyUSB0, 10:25:09

Press CTRL-A Z for help on special keys

 0
U-Boot# echo $serverip
128.247.125.30
U-Boot# editenv serverip
edit: 128.247.125.31
U-Boot# boot
link up on port 0, speed 100, full duplex
BOOTP broadcast 1
DHCP client bound to address 128.247.125.167
link up on port 0, speed 100, full duplex
Using cpsw device
File transfer via NFS from server 128.247.125.31; our IP address is 128.247.1257
Filename '/home/train/ti-processor-sdk-linux-am335x-evm-01.00.00.00/targetNFS//.
Load address: 0x82000000
Loading: T T T T T

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB0
```

If you see this screen, you will need to go back and re-read the lab steps to find and correct your error. Common errors include a misspelling in your /etc/exports file, an

incorrect ip address in the U-Boot environment variable serverip, or some other misspelled U-Boot environment variable.

8. When the board is done booting, you will be prompted for a login. You should enter the login root, as seen below.



> **NOTE**
>
> If you connect to the board over minicom when the board is already booted, you many need to hit Enter a few time to see the login prompt.

Lab 2: Writing an Out-of-Tree Module

Description

The user will setup a skeleton directory for the remainder of the lab, learn to run a basic Hello World module, and implement that module using macros and module parameters. Solutions to all sections of the lab can be found in the git repository that is set up with the skeleton directory.

Lab Steps

Setup Skeleton Directory

1. Open a new terminal window on your Linux machine. You can leave your minicom window open in the background to be used later.
2. If you have not used git before, you will first need to configure your name and email.

   git config --global user.name "<Your name>"
   git config --global user.email "<Your email>"

3. Clone the repository from git.ti.com/sitara-linux/driver_training into a new directory called Module_Training within your SDK directory. This repository contains skeleton code designed to save you time during the lab.

   git clone git://git.ti.com/sitara-linux/driver_training <SDK path>/Module_Training

4. Navigate to the new Module_Training folder. A git repository should already exist in this folder. Create and switch to a new branch called my_modules.

   git checkout -b "my_modules"

5. Navigate to the hello folder inside the Module_Training folder and open the Makefile. The Makefile is designed to compile your code and build your .ko file when you call make on the host PC. It will install the module in your target filesystem when you call make install. Finally, it will remove all files created during a build when you call make clean.
6. Edit the path of the KDIR variable to point to your Linux kernel sources directory. For instance <Linux kernel sources path> could be replaced with:

   ti-processor-sdk-linux-am335x-evm-02.00.00.00/board-support/linux-4.1.6-ga7db74e/

7. Edit the path of the install directory to copy your .ko file from your current directory to /lib/modules/<version number>-<commit id>/extra on your board. If the extra folder does not already exist, you should create it. For instance the line in your Makefile could be changed to:

   install hello_version.ko $(HOME)/ti-processor-sdk-linux-am335x-evm-02.00.00.00/trainingNFS/lib/modules/4.1.6-ga7db74e/extra

Write a Hello World Module

1. Open the hello_version.c file. Write a simple module that will print "Hello World" upon loading, and "Goodbye World" upon removal. You will need to create init_module() and cleanup_module() functions, both of which will include printk() statements. Do not forget to include linux/module.h.

2. Call **make** to build your module. If your module compiles correctly, the output should look similar to the image below. Do not forget to export your path in this terminal instance or to define your ARCH and CROSS_COMPILE variables.

```
sitara@Linux_Machine:~/ti-processor-sdk-linux-am335x-evm-01.00.00.03/Module_Training/hello$ make ARCH=arm
CROSS_COMPILE=arm-linux-gnueabihf-
make -C /home/sitara/ti-processor-sdk-linux-am335x-evm-01.00.00.03/board-support/linux-4.1-xxx/ M=$PWD mod
ules
make[1]: Entering directory `/home/sitara/ti-processor-sdk-linux-am335x-evm-01.00.00.03/board-support/linu
x-4.1-xxx'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/home/sitara/ti-processor-sdk-linux-am335x-evm-01.00.00.03/board-support/linux
-4.1-xxx'
sitara@Linux_Machine:~/ti-processor-sdk-linux-am335x-evm-01.00.00.03/Module_Training/hello$
```

   If your module did not compile correctly, address any errors or warnings listed before proceeding.

   If you misspelled your path in the Makefile you should see an error that says:

   > No such file or directory. Stop.
   > make: *** [all] Error 2

   If you did not export your path in this terminal instance, you will see an error that says:

   > make[1]: arm-linux-gnueabihf-gcc: Command not found.

   Recall from the previous lab, we used the command:

   > **export PATH=$PATH:/home/<user>/<SDK>/linux-devkit/sysroots/<arago linux>/usr/bin/**

3. When you module compiles with no errors or warnings, call **sudo make install** to copy the .ko file into your target filesystem.
4. Open your minicom window and navigate to /lib/modules/<version number>-<commit id>/extra on the target. You should see the hello_version.ko file that you built and copied over in the previous steps. Load your module using the following command. You should see the "Hello World" statement appear. Ignore any warnings about kernel taints for now.

   insmod hello_version.ko

5. Check that your module is on the list of loaded modules with the **lsmod command**. Your module should also appear in /proc/modules which you can view with the command:

   cat /proc/modules

6. Then use **lsmod** to ensure that your module removes correctly after using the following command:

   rmmod hello_version.ko

7. On your host machine, commit your changes to git.

   git add hello_version.c Makefile
   git commit -m "My first module"

Implement Macros

1. Edit your module to use the __init and __exit macros.
2. Edit your module to use the module_init() and module_exit() macros to rename the initialize and remove functions to hello_module_init and hello_module_exit respectively. Do not forget to include linux/init.h. Ensure that your module loads and removes correctly.
3. Edit your module to display licensing information, your name, and a description of the module when you run modinfo helloversion.ko on your host machine using the following macros. The GPL license will also prevent kernel taint warnings from appearing in the future.

   <syntaxhighlight lang="c"> MODULE_LICENSE("GPL"); MODULE_AUTHOR("<Your name>"); MODULE_DESCRIPTION("Hello World"); </syntaxhighlight>

4. Ensure that your module still loads and removes properly, and then commit your changes to git.

   git add hello_version.c
   git commit -m "Module using macros"

Use Module Parameters

1. Edit your module to to take in a parameter who from the user, so that your module will say "Hello <who>" and "Goodbye <who>" when loaded and removed. You should declare a global who variable, use module_param(), and edit your printk() statements.

   static char *who = "World";
   module_param(who, charp, 0);

2. Ensure that your module outputs "Hello <Your name>" and "Goodbye <Your name>" instead of "Hello World" and "Goodbye World" when loaded and removed with the following commands:

   insmod hello_version.ko who="<Your name>"
   rmmod hello_version.ko

3. Commit your changes to git.

   git add hello_version.c
   git commit -m "Module with IO"

Create a Runtime Sensor

1. Edit your module to display the number of seconds that the module was running when you remove it. You will need to create 2 global timeval structures and pass them as a pointers to the do_gettimeofday() function to populate them, then reference the tv_sec field of the structures for the time in seconds. Use the Linux Cross Reference (http://lxr.free-electrons.com/) to identify the necessary include statement.

   <syntaxhighlight lang="c"> struct timeval tstart; int time; do_gettimeofday(&tstart); time = tstart.tv_sec; </syntaxhighlight>

2. When your module is functioning correctly, commit the changes to your git repository.

   git add hello_version.c
   git commit -m "Runtime module"

3. The solutions to all sections of lab 2 can be found in the repository you cloned at the beginning of the lab under Module_Training/Module_Solutions/Lab2. Compare your module to the solution module to ensure that you understood the lab correctly.

Lab 3: Debugging Methods and Tools

Description

Lab Steps

Changing Priority of Console

1. Connect to your board over minicom. Clear the screen and check the current log level with the following command. If you do not clear the screen first, the values will often be illegible.

   clear
   cat /proc/sys/kernel/printk

   The default settings should be 7 4 1 7, referring to the current console log level, the default output level, the minimum log level, and the boot-time default log level respectively.

2. On your host PC, navigate to the debugging folder inside of the Module_Training folder on the host PC. Edit the Makefile to reference the correct paths, just as you did in the previous lab. Open drvbroken.c and read through it. Ensure that you understand what it is supposed to do.
3. Make sure that you have exported your PATH variable in this terminal instance, and call **make** and **sudo make install** to compile drvbroken.c into a .ko file and copy it over to trainingNFS. Be sure to use the correct make command.
4. Load the module and observe which lines appear in the console and which ones do not. Then remove the module.
5. As seen earlier, the current console log level is 7. Change the log level to 4 with the following command:

   dmesg -n 4

6. Reload the module and observe which lines no longer appear. Remove your module.
7. Change the priority of the printk() statement in the module. Recompile and reload your module each time you change the priority. Observe the statement appear and disappear as you change the priority.

   > **NOTE**
   >
   > You may have noticed that log level 7 messages and pr_debug() commands do not appear despite the console's priority level. Enabling these messages requires compiling the kernel with dynamic debugging. For most debugging purposes though, you can simply use printk(), which defaults to log level 4, or "warning," if no priority is specified.

8. Change the log level of the console back to 7.

Lab 4: I2C Nunchuk Module

Description

During this lab, you will wire a Nunchuk device to the board, create an I2C bus to recognize the device, and implement a basic I2C driver.

Lab Steps

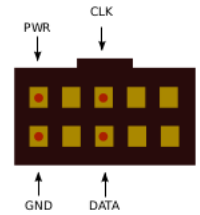
Setup Nunchuk

1. Documentation on the nunchuk can be found here (http://web.engr.oregonstate.edu/~sullivae/ece375/pdf/nunchuk.pdf).

   - Take note of the unique I2C address, 52, and the frequency used for communication, 100kHz, for later in the lab.
   - The nunchuk itself has 6 pins as seen in the diagram below (taken from nunchuk documentation listed above).

Figure 1: Wii Nunchuk Pinout

   - The UEXT Connector breaks out these pins as seen in the image below:



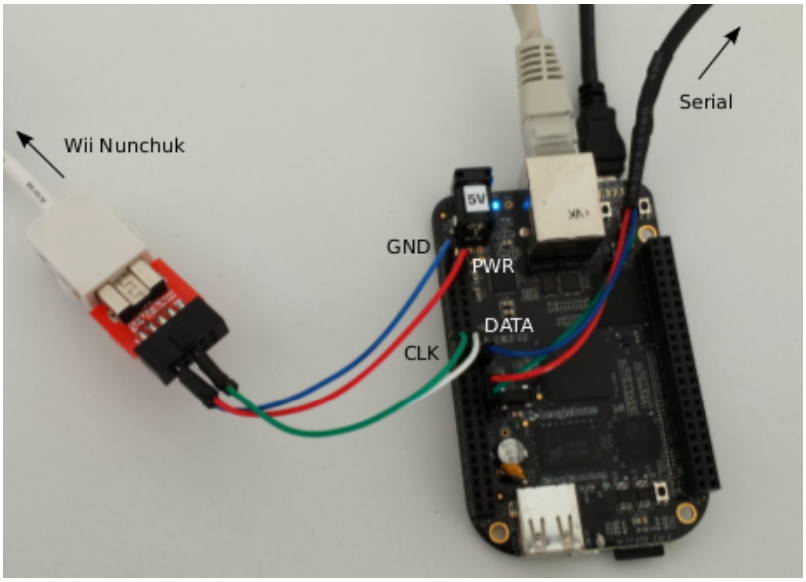
Nunchuk i2c pinout
(UEXT connector from Olimex)

- We will connect the nunchuk to the second I2C port of the CPU (i2c1). The pins for i2c1 are available on the P9 connector of your BeagleBone Black board.
- Information on P9 connector can be found in the BeagleBone Black documentation (https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true). Click on Connector P9 under section 7.1.2 in the table of contents to see the pinout of the P9 connector.
- If you recall, I2C only requires 2 wires: SDA for data and SCL for a clock signal. If you look in the P9 connector pinout, you can see that SCL for I2C1 is pin 17 and that SDA for I2C1 is pin 18, as show in the snapshot of table 13 below. You can also see that ground can be found on pins 1 or 2 and that 3.3V can be found on pins 3 or 4.

PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE
1,2						GND	
3,4						DC_3.3V	
5,6						VDD_5V	
7,8						SYS_5V	
9						PWR_BUT	
10	A10					SYS_RESETn	
11	T17	UART4_RXD	gpmc_wait0	mii2_crs	gpmc_csn4	rmii2_crs_dv	mmc1_s
12	U18	GPIO1_28	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_
13	U17	UART4_TXD	gpmc_wpn	mii2_rxerr	gpmc_csn5	rmii2_rxerr	mmc2_s
14	U14	EHRPWM1A	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a
15	R13	GPIO1_16	gpmc_a0	gmii2_txen	rmii2_tctl	mii2_txen	gpmc_a
16	T14	EHRPWM1B	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a
17	A16	I2C1_SCL	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci	pr1_uart0
18	B16	I2C1_SDA	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone	pr1_uart0
19	D17	I2C2_SCL	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs
20	D18	I2C2_SDA	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs
21	B17	UART2_TXD	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B	pr1_uart0
22	A17	UART2_RXD	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A	pr1_uart0
23	V14	GPIO1_17	gpmc_a1	gmii2_rxdv	rgmii2_rxdv	mmc2_dat0	gpmc_a

2. Now connect the nunchuk pins:

- GND pin on the nunchuk UNEXT connector to P9 pins 1 or 2 (GND)
- PWR pin on the nunchuk UNEXT connector to P9 pins 3 or 4 (DC_3.3V)
- CLK pin on the nunchuk UNEXT connector to P9 pin 17 (I2C1_SCL)
- DATA pin on the nunchuk UNEXT connector to P9 pin 18 (I2C1_SDA)



Declare a Second I2C Bus

1. Navigate to <SDK path>/board-support/linux-<version number>-<commit id>/arch/arm/boot/dts
2. Open the device tree file used by the BeagleBone Black, am335x-boneblack.dts. You can see at the top that this file includes 2 files

include “am33xx.dtsi”

include “am335x-bone-common.dtsi”

3. Open both of these files.
4. If you look through am33xx.dtsi, you will find the declaration of three I2C controllers, including i2c1. You will also see that i2c1 is currently disabled.

<syntaxhighlight lang="c"> i2c1: i2c@4802a000 {

```
                compatible = "ti,omap4-i2c";
                address-cells = <1>;
                size-cells = <0>;
                ti,hwmods = "i2c2";
                reg = <0x4802a000 0x1000>;
                interrupts = <71>;
                status = "disabled";
};
```

</syntaxhighlight>

5. You will now need to declare a new I2C bus and enable it. Look through am335x-bone-common.dtsi for the node of the first I2C bus, i2c0. It should look something like:

<syntaxhighlight lang="c"> &i2c0 {

```
  <Properties>
```

}; </syntaxhighlight>

6. Just below the node of i2c0, add a new node declaring a second I2C bus, i2c1. Set the status to okay to enable it.
7. If you recall when reading the nunchuk documentation, the nunchuk communicates at a frequency of 100kHz so set the clock-frequency property to 100000. Look at the node of i2c0 if you are confused about formatting. We will add the properties pinctrl-names and pinctrl-0 later.

Declare Nunchuk Device

1. Add a child node into your new bus corresponding to the nunchuk device. If you recall when reading the nunchuk documentation, the nunchuk has an I2C address of 52. Your device should also have a compatible property indicating that Nintendo manufactured the device and that the device is called a nunchuk.

<syntaxhighlight lang="c"> nunchuk: nunchuk@52 { compatible = “nintendo,nunchuk”; reg = <0x52>; }; </syntaxhighlight>

Recompile Device Tree

1. Navigate to the root of your Linux kernel sources. If you recall from lab 1, you can recompile your device tree (.dtb file) by first exporting your PATH variable in this terminal instance and then using the make command.

export PATH=$PATH:<SDK path>/linux-devkit/sysroots/i686-arago-linux/usr/bin/
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- am335x-boneblack.dtb

2. Replace your old device tree with the newly compiled version by copying the am335x-boneblack.dtb file to the <SDK path>/trainingNFS/boot directory.
3. Reboot your board.
4. If you edited the device tree properly, you should be able navigate to /proc/device-tree/ocp/i2c\@4802a000 in minicom and see your new nunchuk node nunchuk@52.

Implement Basic I2C Driver

1. On the host PC, navigate to the nunchuk folder inside the Module_Training folder that you downloaded in lab 2. Edit the Makefile so that it functions correctly. Refer to the Setup Skeleton Directory section of lab 2 if necessary.
2. Open nunchuk.c. Write a static probe function called nunchuk_probe() that prints the message “Hello Nunchuk” when called and returns the int 0. The function should take in two parameters: a pointer to an i2c_client structure called client and a pointer to a constant i2c_device_id structure called id.

<syntaxhighlight lang="c"> static int nunchuk_probe(struct i2c_client *client, const struct i2c_device_id *id) </syntaxhighlight>

3. Write a static remove function called nunchuk_remove() that prints the message "Goodbye Nunchuk" when called and returns the int 0. The function should take in a pointer to an i2c_client structure called client.

<syntaxhighlight lang="c"> static int nunchuk_remove(struct i2c_client *client) </syntaxhighlight>

4. Next create a constant, static array of of_device_id structures called nunchuk_dt_ids. It should contain one structure with a compatible field set to "nintendo,nunchuk". The compatible string here must be exactly the same as the compatible string of the nunchuk child node in the I2C bus node in order to match new nunchuk devices to this driver. You will also need to use the MODULE_DEVICE_TABLE() macro.

<syntaxhighlight lang="c"> static const struct of_device_id nunchuk_dt_ids[] = {

```
  { .compatible = "nintendo,nunchuk", },
  {}
```

}; MODULE_DEVICE_TABLE(of, nunchuk_dt_ids); </syntaxhighlight>

5. Create a constant, static array of i2c_driver structures called nunchuk_driver. It should contain one structure with the name and driver_data fields set to "nunchuk" and the int 0 respectively. You will also need to use the MODULE_DEVICE_TABLE() macro.

<syntaxhighlight lang="c"> static const struct i2c_device_id nunchuk_id[] = {

```
  { "nunchuk", 0 },
  {}
```

}; MODULE_DEVICE_TABLE(i2c, nunchuk_id); </syntaxhighlight>

6. Below both of your new functions and both of your new structures, you can now define the driver itself in a static i2c_driver structure. Call the structure nunchuk_driver(). This function should have its probe and remove fields set to the probe and remove functions you wrote earlier. The id_table field should be set to the device identifier array, nunchuk_id(), which you wrote in the previous step. The driver field should be set to a new structure, with the fields set as shown below.

<syntaxhighlight lang="c"> static struct i2c_driver nunchuk_driver = {

```
.probe = nunchuk_probe,
.remove = nunchuk_remove,
.id_table = nunchuk_id,
.driver = {
      .name = "nunchuk",
      .owner = THIS_MODULE,
      .of_match_table = of_match_ptr(nunchuk_dt_ids)
}
```

}; </syntaxhighlight>

7. Finally, register the driver to the I2C bus with the macro module_i2c_driver().

   <syntaxhighlight lang="c"> module_i2c_driver(nunchuk_driver); </syntaxhighlight>

8. Use the **make** and **sudo make install** commands to compile and copy your .ko module to the correct location. If there are any errors or warnings, address them before moving on to the next step. Remember you must first export your PATH variable before the make command will work.

Locate Device and Driver in /sys

1. Open your minicom window and load and remove your module. Ensure that the "Hello Nunchuk" and "Goodbye Nunchuk" messages appear as they should.
2. If your module is working correctly, you should also be able to find a representation of both your device and your driver in /sys. The nunchuk device can be found under /sys/bus/i2c/devices/i2c-1/1-0052. You can cat the contents of the name file in this folder to see that the device is named nunchuk. The driver can be found under /sys/bus/i2c/drivers/nunchuk. The nunchuk driver representation will only appear when the nunchuk module is inserted into the kernel. You should also be able to find another representation of the device within this folder.
3. When your module is working properly, commit your changes to git.

   git add nunchuk.c Makefile
   git commit -m "Hello Nunchuk"

4. The solutions to all sections of lab 4 can be found in the repository you cloned in lab 2 under Module_Training/Solutions/Lab4. Compare your module to the solution module to ensure that you understood the lab correctly.

Lab 5: Pin Muxing and I/O with the Nunchuk

Description

In this lab, the user will use pin muxing to allow the nunchuk device to communicate with the board, add pinctrl properties to the device tree, and write functions to initialize and read data from the device.

Lab Steps

Pin Muxing Configuration

> If you recall, our nunchuk and module are set up to use the bus i2c1. In order for the module to communicate with the nunchuk over the bus, we must first set up pin muxing. Therefore, we must look up the pin muxing configuration data from the BBB documentation.

1. I2C requires 2 wires: SDA for data and SCL for a clock signal. In the last lab, you located these two functionalities, I2C1_SCL and I2C_SDA, in the BBB System Reference Manual (https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true) under Connectors -> Connector P9 -> Expansion Header P9 Pinout. Locate these functionalities again, but this time use Table 13 to identify the mode and SoC pins (under the column PROC) used by these functionalities.

PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE
1,2						GND	
3,4						DC_3.3V	
5,6						VDD_5V	
7,8						SYS_5V	
9						PWR_BUT	
10	A10					SYS_RESETn	
11	T17	UART4_RXD	gpmc_wait0	mii2_crs	gpmc_csn4	rmii2_crs_dv	mmc1_s
12	U18	GPIO1_28	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_
13	U17	UART4_TXD	gpmc_wpn	mii2_rxerr	gpmc_csn5	rmii2_rxerr	mmc2_s
14	U14	EHRPWM1A	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a
15	R13	GPIO1_16	gpmc_a0	gmii2_txen	rmii2_tctl	mii2_txen	gpmc_a
16	T14	EHRPWM1B	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a
17	A16	I2C1_SCL	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci	pr1_uart0
18	B16	I2C1_SDA	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone	pr1_uart0
19	D17	I2C2_SCL	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs
20	D18	I2C2_SDA	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs
21	B17	UART2_TXD	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B	pr1_uart0_
22	A17	UART2_RXD	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A	pr1_uart0_

2. Now that you know the SoC pins (A16 and B16) and the mode number (2), open the CPU datasheet (http://www.ti.com/lit/ds/symlink/am3359.pdf). Locate the ZCZ Pin Map under the Pin Diagrams section. Make sure you are not looking at the ZCE package information instead!
3. Use this table to look up information on the SoC pins, as shown below. Take note of the SoC pin names. Confirm the mode of the I2C1_SCL and I2C1_SDA functionalities. Also confirm that each SoC pin supports pull-up mode, which is necessary for I2C to function properly.

ZCZ Pin Map [Section Left - Top View]

	A	B	C	D	E	F
18	VSS	EXTINTn	ECAP0_IN_PWM0_OUT	UART1_CTSn	UART0_CTSn	MMC0_DAT2
17	SPI0_SCLK	SPI0_D0	I2C0_SDA	UART1_RTSn	UART0_RTSn	MMC0_DAT3
16	SPI0_CS0	SPI0_D1	I2C0_SCL	UART1_RXD	UART0_TXD	USB0_DRVVBUS
15	XDMA_EVENT_INTR0	PWRONRSTn	SPI0_CS1	UART1_TXD	UART0_RXD	USB1_DRVVBUS
14	MCASP0_AHCLKX	EMU1	EMU0	XDMA_EVENT_INTR1	VDDS	VDDSHV6
13	MCASP0_ACLKX	MCASP0_FSX	MCASP0_FSR	MCASP0_AXR1	VDDSHV6	VDD_MPU
12	TCK	MCASP0_ACLKR	MCASP0_AHCLKR	MCASP0_AXR0	VDDSHV6	VDD_MPU
11	TDO	TDI	TMS	CAP_VDD_SRAM_MPU	VDDSHV6	VDD_MPU
10	WARMRSTn	TRSTn	CAP_VBB_MPU	VDDS_SRAM_MPU_BB	VDDSHV6	VDD_MPU
9	VREFN	VREFP	AIN7	CAP_VDD_SRAM_CORE	VDDS_SRAM_CORE_BG	VDDS
8	AIN6	AIN5	AIN4	VDDA_ADC	VSSA_ADC	VSS
7	AIN3	AIN2	AIN1	VDDS_RTC	VDDS_PLL_DDR	VDD_CORE
6	RTC_XTALIN	AIN0	PMIC_POWER_EN	CAP_VDD_RTC	VDDS	VDD_CORE

4. Finally, open the BBB Technical Reference Manual (http://www.ti.com/lit/ug/spruh73l/spruh73l.pdf) and use the SoC pin names you just identified (SPI0_CS0 and SPI0_D1) to look up the offsets for the registers controlling each of these pins. This information can be found under CONTROL_MODULES Registers table, as shown below.

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
948h	conf_mdio		Section 9.3.1.49
94Ch	conf_mdc		Section 9.3.1.49
950h	conf_spi0_sclk		Section 9.3.1.49
954h	conf_spi0_d0		Section 9.3.1.49
958h	conf_spi0_d1		Section 9.3.1.49
95Ch	conf_spi0_cs0		Section 9.3.1.49
960h	conf_spi0_cs1		Section 9.3.1.49
964h	conf_ecap0_in_pwm0_out		Section 9.3.1.49

Add pinctrl Properties to Device Tree

1. Navigate to <SDK path>/board-support/linux-<version number>-<commit id>/arch/arm/boot/dts and open am33xx.dtsi and am335x-bone-common.dtsi, the two .dtsi files used by the BeagleBone Black.
2. Look through am33xx.dtsi for the declaration of the main pinctrl device, am33xx_pinmux. Note the base address of 0x800, which is different from the base address provided in the TRM, where we got the offset values for our SoC pins. Subtract 0x800 from both of the offset values to account for this difference. You should now have the offset values 0x158 and 0x15c.
3. Look through am335x-bone-common.dtsi for the declaration of i2c0_pins, the pinctrl configuration for i2c0. Just below this node, add a new node called pinmux_i2c1_pins, alias i2c1_pins, for the pinctrl configuration of i2c1. Use the pinctrl-single driver to set both of the offset registers to pull-up mode and mux mode 2, as you determined you should from the documentation.

   <syntaxhighlight lang="c"> i2c1_pins: pinmux_i2c1_pins { pinctrl-single,pins = < 0x158 (PIN_INPUT_PULLUP | MUX_MODE2) /* spi0_d1.i2c1_sda */ 0x15c (PIN_INPUT_PULLUP | MUX_MODE2) /* spi0_cs0.i2c1_scl */ >; }; </syntaxhighlight>

4. Scroll down and add the pinctrl properties pinctrl-0 and pinctrl-name to your i2c1 bus node so that the bus knows which pin configuration it needs to use.

   <syntaxhighlight lang="c"> pinctrl-names = "default"; pinctrl-0 = <&i2c1_pins>; </syntaxhighlight>

5. Recompile your device tree, copy the zImage and .dtb files as before, and reboot your board.

Test the Pin Muxing

1. If you set up your pin muxing correctly, the using the following command in your minicom terminal will scan the i2c1 bus for devices, which will allow you to see your nunchuk device at address 0x52, as seen below. You will need to select y to proceed when the warning prompt appears.

i2cdetect -r 1

```
root@am335x-evm:~# i2cdetect -r 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- 52 -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
root@am335x-evm:~#
```

If your nunchuk does not appear, check the recent pin muxing configuration changes you made to your am225x-bone-common.dtsi file and verify that the wiring between your nunchuk and the BeagleBone Black is correct.

Device Initialization

Now that communication with the nunchuk has been set up, the first thing the module probe function should do is initialize the nunchuk device.

1. On your host machine, navigate to the nunchuk folder inside the Module_Training folder that you downloaded in lab 2 and open nunchuk.c.
2. In the nunchuk documentation (http://web.engr.oregonstate.edu/~sullivae/ece375/pdf/nunchuk.pdf), you will see that in order to initialize the nunchuk, you need to send the bytes 0xf0 and 0x55 together, followed by the bytes 0xfb and 0x00 together.
3. In order to send bytes to the nunchuk you should first implement a nunchuk_write_registers() function above your probe function. This function should take in a pointer to an i2c_client structure called client, a pointer to a constant character called buf, and an int called count. It should return void.

   <syntaxhighlight lang="c"> void nunchuk_write_registers(struct i2c_client *client, const char *buf, int count) </syntaxhighlight>

4. nunchuk_write_registers() should write the bytes in buf to the nunchuk client using the int i2c_master_send(struct i2c_client *client, const char *buf, int count) function. The call to i2c_master_send() should be followed by a 1 ms delay, using the udelay() function. Use the Linux Cross Reference (http://lxr.free-electrons.com/) to identify the necessary include statements.
5. In your probe function, create two const char arrays. One should contain the bytes 0xf0 and 0x55 and the other should contain the bytes 0xfb and 0x00. Make two calls to your function nunchuk_write_registers() to write both of these arrays respectively to the nunchuk. This will initialize the nunchuk device. Keep in mind that a C array will already be a pointer to the first element in the array. You can get the size of the arrays with the sizeof() function.
6. Recompile and load your module until there are no errors or warnings and the print statements appear as expected.
7. Commit your changes to git.

   git add nunchuk.c
   git commit -m "Initialize Nunchuk"

Read Data from Nunchuk

1. Now that the probe function initializes the nunchuk, you should be able to read data from the nunchuk. Create a new function called nunchuk_read_registers() above your probe function. nunchuk_read_registers() should take in the same parameters as nunchuk_write_registers() except buf should no longer be const. nunchuk_write_registers() should also return void.
2. The first thing you should do is place a 10ms delay at the beginning of the function using the mdelay(0 function. This delay will separate the following I2C action from any previous I2C action.
3. If you look through the nunchuk documentation, you will see that each time you want to read from the nunchuk device, you must first send the byte 0x00. The nunchuk will then return 6 bytes of data. Therefore, the next thing your new nunchuk_read_registers() function should do is send the 0x00 byte with your nunchuk_write_registers() function. This action should be immediately followed by a 10ms delay using the mdelay() function.

   > **NOTE**
   >
   > The nunchuk_write_registers() function already waits 1ms, so you only need to wait 9ms in your nunchuk_read_registers() function). This delay will separate the write I2C action from the read I2C action to follow.

4. nunchuk_read_registers() should read count bytes of data from the nunchuk client and store them in buf using the int i2c_master_recv(struct i2c_client *client, const char *buf, int count) function.
5. You should also write an if-else-statement that prints either an error message or the number of bytes read each time this function is called.
6. Place a call into your probe function to read 6 bytes of data from the nunchuk with your new nunchuk_read_registers() function.
7. At this point, it is important to note an unusual behavior of the nunchuk: the nunchuk will only update its internal register once the register has been read. Therefore in order to read the current state of the nunchuk, you will need to place a second call to your nunchuk_read_registers() function directly below the first.
8. Print the data that you just read from the nunchuk. You will need to print each index of the function individually so it is easier to place the call in a for-loop, as shown below. Each index should print a hex value where buf is the character array you placed the data in.

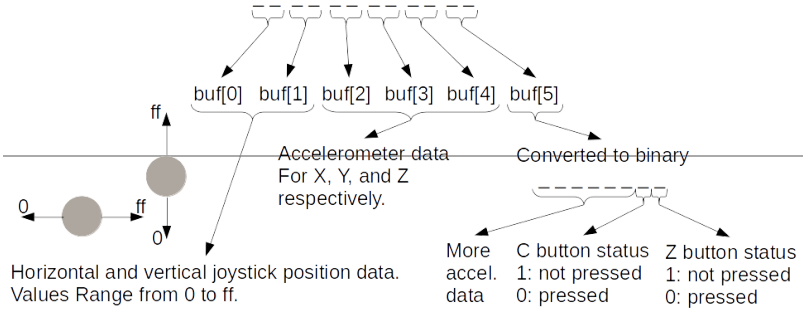
   <syntaxhighlight lang="c"> unsigned int i; for(i=0; i<sizeof(buf); i++) { printk("%x ", buf[i]); } printk("\n"); </syntaxhighlight>

9. Compile and load your module and ensure that there are no errors or warnings and that all print statements appears as expected. At this point, your module should write 2 bytes, write 2 bytes, write 1 byte, read 6 bytes, write 1 byte, read 6 bytes, and then print the current state of the nunchuk register.
10. Then commit your changes to git.

    <syntaxhighlight lang="c"> git add nunchuk.c git commit -m "Read Nunchuk Data" </syntaxhighlight>

Understanding the Data

The meanings behind each of the 6 bytes you just printed can be found in the nunchuk documentation (http://web.engr.oregonstate.edu/~sullivae/ece375/pdf/nunchuk.pdf). As shown below, to view the status of buttons C and Z, we will look at the bottom two bits of the 6th byte.



Check the Button Status

1. Comment out the number of bytes written statement, the number of bytes read statement, and the loop in your probe function that you used to print the register. These statements are no longer necessary and will only clutter your screen.
2. Write two new functions called zPressed() and cPressed() above your probe function. Both of these functions should take in a pointer to a character array, char *buf (this should be the data you read from the nunchuk) and return a boolean, (true when either the z or c button is pressed, false otherwise). These functions should be implemented with boolean operators.

3. Place a call to both the cPressed() and zPressed() functions inside of your probe function. Create if statements that will print 'The C button is pressed!' or 'The Z button is pressed!' if the functions ever return true.
4. Compile and load your module. If it is working correctly, when you load your module while holding either the c or z buttons down, a message will appear in the console.

```
git add nunchuk.c
git commit -m "Button notifier"
```

5. The solutions to all sections of lab 5 can be found in the repository you cloned during lab 2 under Module_Training/Solutions/Lab5. Compare your module to the solution module to ensure that you understood the lab correctly.

Lab 6: Polling and Device Registration

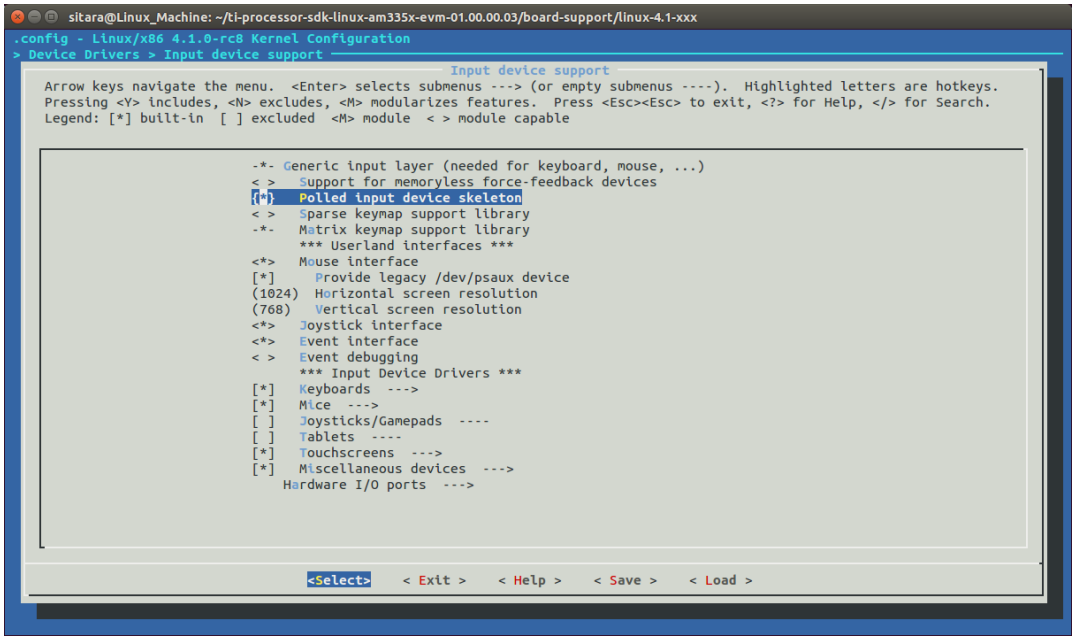
Description

If you recall from the previous lab, you had to load and remove the module each time you wanted to see the updated nunchuk state. During this lab, you will create an interface that will continually look for updates in the nunchuk state. The nunchuk does not have interrupts to notify the I2C master that its state has changed. Therefore, the only way to access device data and detect changes is to regularly poll its registers.

Lab Steps

Recompile the Kernel to Support Polling

1. Rebuild your kernel with the options CONFIG_INPUT_POLLDEV=y, or static support for polled input devices. If you do not include this configuration, you will receive the error could not insert module example.ko: Unknown symbol in module when you attempt to allocate or register the polling device. This option can be enabled in menuconfig which can be accessed with the command make menuconfig from the root of your Linux kernel sources. Enable the option Polled input device skeleton under the sections Device Drivers -> Input device support -> Generic input layer.



Create Logical Device Structure

1. Comment out both calls to nunchuk_read_registers() and the calls to cPressed() and zPressed() and the if/printk-statements just below them in your probe functions. Also comment out any variables no longer in use.
2. Above your nunchuk probe routine, create a new structure called nunchuk_dev. It should contain one field right now, struct i2c_client *12c_client.

<syntaxhighlight lang="c"> struct nunchuk_dev {

```
struct i2c_client *i2c_client;
```

}; </syntaxhighlight>

3. Create a pointer to an instance of this structure in your probe routine called nunchuk. You will need to allocate memory for it using the function devm_kzalloc(). You should also add an if-statement to check that the allocation succeeds and to end the function if it does not.

<syntaxhighlight lang="c"> nunchuk = devm_kzalloc(&client->dev, sizeof(struct nunchuk_dev), GFP_KERNEL); if (!nunchuk) { dev_err(&client->dev, "Failed to allocate memory for logical device\n"); return -ENOMEM; } </syntaxhighlight>

> **NOTE**
>
> With devm_ functions, each allocation or registration is attached to a device structure. When a device or module is removed, all such allocations or registrations are automatically undone, which allows us to greatly simplify driver code.

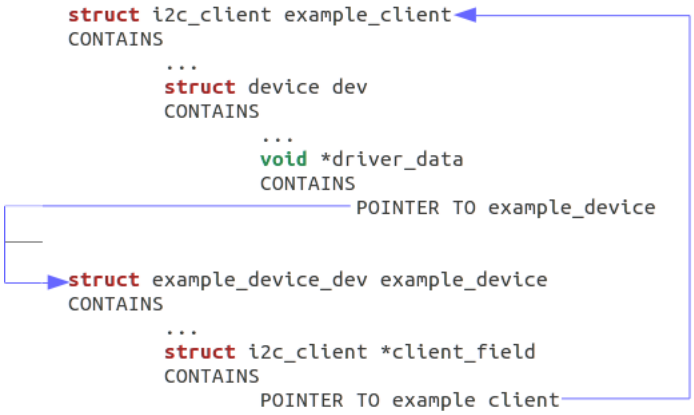
Follow Device Model Conventions

In order for the device model to function properly, our logical and physical device structures need to be linked together. This can be achieved by placing pointers to one another inside of the structures. Since there are no global variables involved in this method of linking the devices, this convention allow the driver to support multiple nunchuk devices.

1. In your probe function, link the logical and physical device structures with pointers.

   <syntaxhighlight lang="c"> nunchuk->i2c_client = client; i2c_set_clientdata(client, nunchuk); </syntaxhighlight>

   Your structures should now contain pointers to one another as shown below.



Create Polling Device

1. Declare a pointer to an input_polled_dev structure in your probe routine called polled_input. You will need to allocate memory for it using the input_allocate_polled_device() function. You should also add an if-statement to return an error if the allocation fails. Search the Linux Cross Reference (http://lxr.free-electrons.com/) for the necessary include statement.

   <syntaxhighlight lang="c"> polled_input = input_allocate_polled_device(); if (!polled_input) {

   ```
   pr_err("Failed to allocate memory for polling device\n");
   ```

   return -ENOMEM; } </syntaxhighlight>

2. You should set the poll interval to 50ms. You should also set the polling routine to a function called nunchuk_poll, which we will implement at a later section of this lab.

   <syntaxhighlight lang="c"> polled_input->poll_interval=50; polled_input->poll=nunchuk_poll; </syntaxhighlight>

3. Create a pointer named input and point it to the struct input_dev field of your input_polled_dev structure. This will make it easier to reference the input device associated with your input_polled_dev structure as we configure it.

   <syntaxhighlight lang="c"> struct input_dev *input = polled_input->input; </syntaxhighlight>

4. In order to configure the input device, you need to set the name, bustype, evbit, and keybit of the structure, as shown below:

   <syntaxhighlight lang="c"> input->name = "Wii Nunchuk"; input->id.bustype = BUS_I2C; set_bit(EV_KEY, input->evbit); set_bit(BTN_C, input->keybit); set_bit(BTN_Z, input->keybit); </syntaxhighlight>

Follow Device Model Conventions

1. Create a new field in your nunchuk_dev structure to connect the logical device to the input device.

   <syntaxhighlight lang="c"> struct input_polled_dev *polled_input; </syntaxhighlight>

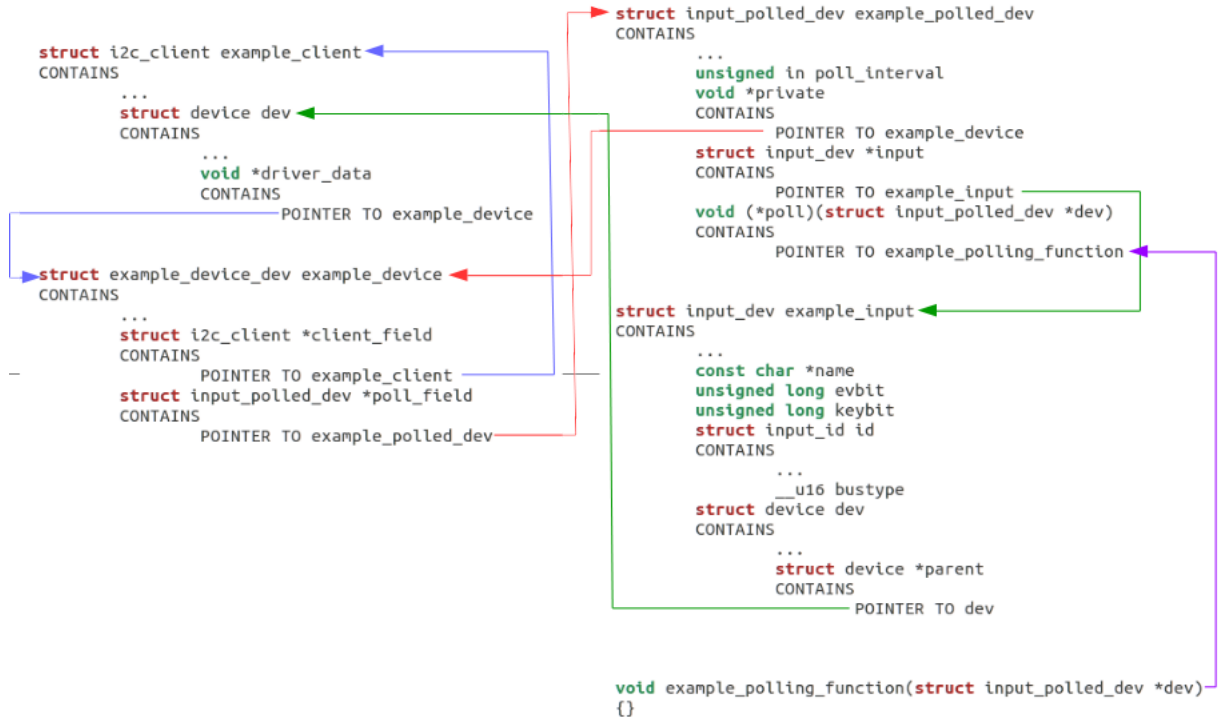
2. In your probe function, point the polled_input field of nunchuk to the polled_input structure your created. Then connect the polled input device to the logical device by setting the private field of polled_input to point to nunchuk.

   <syntaxhighlight lang="c"> nunchuk->polled_input = polled_input; polled_input->private = nunchuk; </syntaxhighlight>

3. Finally, connect the polled input device to the physical device by setting the parent field of the dev structure in input (which is contained within polled_input).

   <syntaxhighlight lang="c"> input->dev.parent = &client->dev; </syntaxhighlight>

4. Your structures should now contain pointers to one another as shown below.



Register Polling Device

1. You should register the polling device in your probe function with the function input_register_polled_device(). You should also add an if-statement that will free the allocated memory of polled_input and return an error if the registration fails. If you do not free the memory before returning an error, you will create memory leaks in the kernel. Be sure to declare the int result.

   <syntaxhighlight lang="c"> if(result<0) { input_free_polled_device(polled_input); pr_err("Failed to register polled device\n"); return -ENOMEM; } </syntaxhighlight>

Implement the Polling Routine

1. Above your probe routine, create a function called nunchuk_poll (the same name you used earlier in the lab). If you look at the definition of the input_polled_dev structure, you will see that the function needs to return void and take in an input_polled_dev structure as a parameter.

   <syntaxhighlight lang="c"> void nunchuk_poll(struct input_polled_dev *polled_input) </syntaxhighlight>

2. By following device model conventions earlier in the lab, you can use the nunchuk_dev structure to retrieve a pointer to the physical device (i2c_client) from the private field of the parameter polled_input.

   <syntaxhighlight lang="c"> struct nunchuk_dev *nunchuk = polled_input->private; struct i2c_client *client = nunchuk->i2c_client; </syntaxhighlight>

3. Declare a 6 character char array called buf and call your nunchuk_read_registers() function (which you created during the last lab) to read from the physical device (in this example, client) and store any data in buf.
4. Use the zPressed() and cPressed() functions you wrote during the last lab to determine if the z or c buttons are currently pressed. Store the output of these functions in two boolean variables called zPress and cPress. If either of them is pressed, use the input_event() function as shown below to notify the input subsystem. You will then need to call input_sync() as there could be two input events at once. Search the Linux Cross Reference (http://lxr.free-electrons.com/) for the necessary include.

   <syntaxhighlight lang="c"> input_event(polled_input->input, EV_KEY, BTN_Z, zPress); input_event(polled_input->input, EV_KEY, BTN_C, cPress); input_sync(polled_input->input);
   </syntaxhighlight>

Implement the Remove Routine

1. Finally, your remove routine needs to undo anything done in your probe function. Retrieve the logical device (struct nunchuk_dev *nunchuk) from the i2c_client structure with the i2c_get_clientdata() function. Then retrieve the input_polled_dev structure from the polled_input field of nunchuk.
2. Unregister the polled device and free its allocated memory.

   <syntaxhighlight lang="c"> input_unregister_polled_device(polled_input); input_free_polled_device(polled_input); </syntaxhighlight>

Test Your Module

1. Compile your module. If there are any errors or warning, address them before proceeding.
2. Next ensure that your module loads and removes correctly. Currently when loaded, your module should only be printing a 'Hello Nunchuk' line and a line that says 'input: Wii Nunchuk as /devices/platform/ocp/4802a000.i2c/i2c-1/1-0052/input/input0'.
3. On your host machine, look for a file called evtest in the nunchuk folder of the Module_Training directory. Copy the file into your trainingNFS directory. To run the tester from the same directory as your nunchuk module, you may copy it into your lib/modules/<version>/extra folder. Otherwise, you will have to navigate in your minicom window to the directory where you copied the file.
4. If you recall from the lectures, when you registered the device, a node to export information to user space was created. The node created for the nunchuk should be called event0 and is located under /dev/input. evtest requires the location of the node as a parameter, as shown below. evtest is an event tester that will read from this node and print a notification each time there is an event detected from the nunchuk. When you run evtest, you should see notifications each time you press or release a button on the nunchuk.

   ./evtest /dev/input/event0

5. When your module is functioning properly, commit your changes to git.

   git add nunchuk.c

git commit -m "Nunchuk User Interface"

6. The solutions to all sections of lab 6 can be found in the repository you cloned during lab 2 under Module_Training/Solutions/Lab6. Compare your module to the solution module to ensure that you understood the lab correctly.

{{

1. switchcategory:MultiCore=
   - For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
   - For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **Processor SDK Linux Training: Introduction to Device Driver Development** here.

Keystone=
- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **Processor SDK Linux Training: Introduction to Device Driver Development** here.

C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article* ***Processor SDK Linux Training: Introduction to Device Driver Development*** *here.*

For techn please po questions http://e2e Please po comment article ***Pr SDK Linu Introduct Device D Developr***

}}

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors
- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity