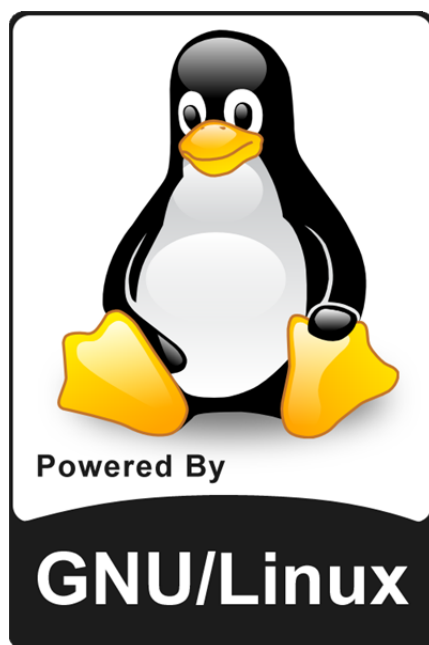


# Introduction to Embedded Linux

---

## *Lab Instructions*



*Revision 3.00  
October 2014*

## Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2012 Texas Instruments Incorporated

## Revision History

August 2012 – Revision 1.0

## Mailing Address

Texas Instruments  
Training Technical Organization  
6550 Chase Oaks Blvd  
Building 2  
Plano, TX 75023

# Lab 01: Booting Linux

---

## Introduction

This lab exercise will demonstrate creating a bootable mini-SD card and using it to boot the Arago Linux distribution (from TI's software development kit) on the AM335x Starter Kit. The SDK contains all files that are needed to boot Linux on the development board.

The lab environment uses a host computer that is running the Ubuntu distribution of Linux. While it is not required to use a Linux-based development PC, doing so is generally more convenient than using a Windows-based environment. Developers who do not have access to a Linux-based development PC may consider setting up a dual-boot environment on their PC so that both Linux and Windows may be booted. Another option is to use virtualization software such as VMware or Virtual Box in order to run a Linux virtual machine within their Windows environment.

## Module Topics

<b>Lab 01: Booting Linux .....</b>	<b>1-1</b>
<i>Module Topics.....</i>	<i>1-2</i>
<i>A. Create a Bootable SD Card.....</i>	<i>1-3</i>
<i>B. Boot Linux on the AM335x Starter Kit .....</i>	<i>1-6</i>

## A. Create a Bootable SD Card

1. **Power on the development computer.**

The development PC used in these lab exercises has Ubuntu 12.04 installed. There is a single user account (username is “user”) which has a null password and a null sudo password. Also, the computer has been configured to automatically log into this user account.

2. **Insert the mini-SD card, with adapter, into the SD/MMC card reader and plug the card reader into the PC USB slot.**

You may see two partitions of the mini-SD card show up on the desktop when the card reader is attached. This is because we do not erase the SD cards between workshops, so that this SD card may already contain all of the files needed to boot. Also, if you are using a new AM335x starter kit out of the box, the micro-SD card that ships with the kit also comes installed with the files necessary to boot Linux.

Don’t worry; we will reformat the card in step 5 to ensure that the lab exercise is completed successfully.

3. **Launch an xterm terminal.**

You can use one of the terminal icons from the desktop. Two terminal icons are provided: one with a black background and one with a white background. This will become important in later labs when we use a terminal to interface to Linux running on the AM335x Starter Kit. It is easy to become confused when multiple terminals are open as to which is accessing Linux on the host PC and which is accessing Linux on the development board. For now you may use either terminal according to preference.

4. **Determine the Small Computer Systems Interface (SCSI) device-node mapping of your SD/MMC card reader.**

Within the xterm terminal window type:

Note: you do not need to type anything before the dollar symbol (“ubuntu\$”). This is being used to designate the terminal prompt.

```
ubuntu$ sudo sg_map -i
```

You should see something similar to the following output to the terminal:

```
/dev/sg0 /dev/scd0 NECVMWar VMware IDE CDR10 1.00
/dev/sg1 /dev/sda  VMware,   VMware Virtual S  1.0
/dev/sg2 /dev/sdb  USB 2.0   SD/MMC Reader
```

In the above example, the SD/MMC Reader has been mapped to the /dev/sdb device node. This is very likely what you will observe as well.

sg\_map is part of the sg3-utils package available through Ubuntu’s Aptitude package manager. It has already been installed for you on this virtual machine, but if you are running these labs on a different Linux computer, you can install the utility by typing:

```
ubuntu$ sudo apt-get install sg3-utils
```

### 5. Unmount the micro-SD card

Within Linux, the format operation cannot be completed on a storage device while it is mounted into the root filesystem. Linux mounts the partitions of a block storage device as separate mount points, so unmount each one:

(This step assumes that the device node determined in step 4 is “sdb,” otherwise replace “sdb” with the correct node.)

```
ubuntu$ sudo umount /dev/sdb1
ubuntu$ sudo umount /dev/sdb2
ubuntu$ sudo umount /dev/sdb3
```

Note: your micro-SD card may or may not have a third partition. If there is no third partition, you will get a warning when you attempt to unmount, but this can be ignored.

### 6. Change into the Software Development Kit (sdk) directory.

```
ubuntu$ cd /home/user/ti-sdk-07.01.00.00
```

Note: You can use the autofill feature to make this faster by typing “cd /home/user/ti-” and then pressing the <tab> key.

### 7. List the contents of this directory.

```
ubuntu$ ls
```

You should see :

bin	directory holding binary (executable) files and scripts for the sdk. The script that we will use to write the bootable sd card is here.
board-support	directory holding source code for the linux kernel and uboot as well as pre-built versions of both. The u-boot and kernel we will place on the sd card are in the “prebuilt-images” subdirectory
docs	sdk documentation
example-applications	example linux applications including the matrix gui application that comes programmed onto Sitara evaluation kits and sdks from the factory
filesystem	the root filesystem that will be installed onto the bootable sd card is located here. It was built (as was the entire sdk) using OpenEmbedded.
host-tools	contains the pin multiplexing (pinmux) utility
linux-devkit	contains the gnu cross-compiler for the am335x and associated libraries as well as qtopia development tools.

### 8. Change into the bin directory of the sdk

```
ubuntu$ cd bin
```

### 9. Execute the “create-sdcard.sh” script, being sure to use root permissions via the “sudo” (“switch user do”) command.

```
ubuntu$ sudo ./create-sdcard.sh
```

This script creates two partitions on the multimedia card and formats the partitions (vfat on partition 1 and ext3 on partition 2). The partitioning requirements needed for a micro-SD card to be bootable on a TI ARM device must be precise, and this script guarantees that the partitioning is done correctly.

The script will ask some questions, the next steps specify what you should answer.

**10. (create-sdcard.sh) Specify device number/node for the micro-SD card.**

The script will list the devices on the system to which the Linux boot files may be written. You should see something similar to:

#	major	minor	size	name
1:	8	88	3813376	sdb

In particular, the “name” listed in the final column should be the same as was determined in step 4. There should be only one option, so enter “1” and continue.

**11. (create-sdcard.sh) If prompted to repartition the sd card, press “y”**

If the script determines that the sd card you are using has already been partitioned, it will ask if you wish to keep the current partitioning or repartition. If asked, press “y” for yes. If not, continue to step 12.

**12. (create-sdcard.sh) Specify two partitions.**

A bootable SD card requires two partitions. The first partition contains the MLO, u-boot and kernel image. The second partition contains the root file system (which must be in a separate partition).

The create-sdcard.sh script provides the option of creating a third partition. This is because the micro-SD card that comes with the AM335x starter kit boards has a third partition that contains the installation files for the SDK and for Code Composer Studio (CCS). We have no need for a third partition to use as general storage, so select the 2-partition option.

**13. (create-sdcard.sh) Enter “y” to continue after partitioning is complete.****14. (create-sdcard.sh) Choose option “1” to install pre-built image from sdk.**

Note that the sdk used in this workshop is not the standard sdk-07.01.00.00 from Texas Instruments. The sdk has been rebuilt using a workshop bitbake overlay layer that provides additional features to the sd card image beyond what is provided in the sdk. More detail on this will be provided throughout the workshop.

**15. Eject the multimedia card.**

```
ubuntu$ sudo eject /dev/sdb
```

For efficiency, Linux rarely writes directly to a device, but instead writes into a RAM buffer, which is then copied to the device. In some cases, the entire buffer may not be written until the device is ejected, which could potentially cause corruption if the micro-SD is pulled from the reader before the device is properly ejected.

The “create-sdcard.sh” script actually ejects the device for you (technically it unmounts the device using the “umount” command for those advanced students who are aware of the difference between the two operations), but it is good practice to get in the habit of always ejecting before removing the micro-SD card just in case it is needed.

**16. Remove the mini-SD card from the SD/MMC card reader and insert it into the mini-SD slot of the AM335x Starter Kit.**

On the AM335x Starter Kit, the metal leads of the micro-SD card should face away from the printed circuit board.

## B. Boot Linux on the AM335x Starter Kit

### 17. Connect an Ethernet cable between the host PC and the AM335x starter kit.

The first Ethernet connection (eth0) on the AM335x starter kit corresponds to the RJ-45 jack that is further from the USB connector (labeled “J6” on the PCB). This is the one you should use.

### 18. Connect a USB cable between the host PC and the AM335x starter kit.

The AM335x starter kit has connections for UART, JTAG (xds-100) and an Ethernet gadget driver over the USB cable. Support for the UART-over-USB functionality is provided by an ASIC from FTDI.

### 19. Power on the AM335x Starter Kit.

There are two buttons on the non-LCD side of the AM335x starter kit. The button which is next to the barrel connector for the power cable is the reset button. The button on the side across from this connector is the power button.

Once you press the power button, you should see a green LED light up on the base board.

### 20. Start minicom.

```
ubuntu$ minicom
```

Minicom has been pre-configured to use the device “/dev/ttyUSB1” which is the device node corresponding to the UART-over-USB driver provided by the FTDI chip. It is configured for 115,200 baud, 8 bit data, no parity, 1 stop bit, and no flow control.

It takes a moment after powering on the AM335x starter kit before the /dev/ttyUSB1 device node is recognized and mounted by the system, so if you receive either of the following messages:

```
minicom: cannot open /dev/ttyUSB1: No such file or directory
or
```

```
minicom: cannot open /dev/ttyUSB1: Device or resource busy
```

wait a few moments and try again.

### 21. Enter user “root” at the login prompt (no password.)

**Note:** you may need to press the “Enter” key in order to see the login prompt.

Upon a successful boot of Arago Linux, you will be presented the login prompt on the minicom console (after quite a bit of feedback as Linux boots):

```
am335x login: root
```



## **22. Determine IP addresses of Ethernet and Ethernet-over-USB connections**

The standard sdk version 07.01.00.00 from Texas Instruments initializes with a mass storage USB gadget driver instead of the Ethernet gadget driver used in the workshop. This is one of the changes made in the workshop's bitbake overlay layer, which will be discussed in module 2.

In the host Ubuntu machine, type the command:

```
ubuntu$ ifconfig
```

You should see three entries – eth0, lo and USB0.

You may likewise use the “ifconfig” command on the AM335x starter kit to test the connections, via the minicom connection that you established in step 20.

```
am335x$ ifconfig
```

You should see the same three connections (with different IP addresses for eth0 and USB0) on the am335x starter kit.

## **23. (If necessary) Manually disconnect and reconnect USB0.**

You should find the Ethernet USB gadget driver to be fairly robust, but if you do not see a USB0 connection upon booting the system, you can often bring the interface up by disconnecting and then reconnecting manually:

```
ubuntu$ sudo ifdown usb0
```

```
ubuntu$ sudo ifup usb0
```

If you still do not see a “USB0” connection, ask your instructor for assistance.

## **24. Test Ethernet and Ethernet-over-USB connections**

(You can press <ctrl-c> while you have scope within the terminal in order to exit the ping utility.)

```
ubuntu$ ping 192.168.1.2
```

```
ubuntu$ ping 192.168.7.2
```

```
am335x$ ping 192.168.1.1
```

```
am335x$ ping 192.168.7.1
```

As you determined in step 22, the gigabit Ethernet connection is on the “192.168.1.x” subnet and the Ethernet-over-USB connection is on the “192.168.7.x” subnet.

In both cases, the Ubuntu host machine has final octet set to 1 and the am335x starter kit has the final octet set to 2. These addresses are set statically in the “/etc/network/interfaces” configuration file of each system.

## **25. View the startup configuration.**

The startup script methodology used by the Arago distribution is called SysV (system five) and is a common startup methodology across various Linux distributions. Much information is available on the internet for those wishing to learn more about SysV.

```
am335x$ ls /etc/rc5.d
```

(Page intentionally left blank)

# Lab 02: Linux Terminal Manipulation

---

## Introduction

Lab 01 consists of two parts. Linux is an inherently text-based operating system, and, particularly in an embedded environment, users will be able to leverage the toolset of Linux most effectively with a strong working knowledge of text-based terminal commands. In the first section you will learn basic terminal manipulation, with commands such as “cd,” “ls,” and “cp,” and introducing a few helpful features along the way such as autofill and history.

The second and third sections cover Linux redirection and Linux shell scripting. These are advanced and very powerful features of the terminal interface, and users who learn these tools will enjoy an added level of control over Linux systems from the terminal interface.

## Module Topics

<b>Lab 02: Linux Terminal Manipulation .....</b>	<b>2-1</b>
<i>Module Topics.....</i>	<i>2-2</i>
<i>1 Basic Terminal Manipulation .....</i>	<i>2-3</i>
<i>2 Linux Redirection.....</i>	<i>2-6</i>
<i>(Advanced) Linux Scripting .....</i>	<i>2-7</i>

# 1 Basic Terminal Manipulation

Note that these exercises are to be completed on the Ubuntu host computer and not on the AM335x starter kit.

1. **Change to your home directory.**

```
ubuntu$ cd ~
```

The tilde “~” character is Linux shorthand for the current user’s home directory, which is located at “/home/<user>”

2. **Print the current directory path.**

```
ubuntu$ pwd
```

The “pwd” command is short for “print working directory.”

3. **List the contents of the current directory.**

```
ubuntu$ ls
```

4. **View the help page for the “mkdir” command.**

```
ubuntu$ mkdir --help
```

Most commands support the “--help” option, which prints a help listing for the command. Many commands also support a shorthand version, “-h” as well.

5. **Make a new directory named “lab02.”**

```
ubuntu$ mkdir lab02
```

6. **Change into the lab02 directory.**

7. **Create a new, empty file named “myhugelongfilename.”**

There are a number of ways to create a new file, but the simplest is by using the “touch” command. The “touch” command updates the timestamp on a file if it exists and, if the file does not exist, creates it.

```
ubuntu$ touch myhugelongfilename1
```

8. **List “myhugelongfilename1” specifically, i.e. without listing any other files, using Linux autofill.**

The autofill capability in the Linux terminal is extremely helpful. Linux will autofill the name of a file or directory in the current directory when you press the “tab” key. You can try this with the “ls” command:

Note: this is the letter “m” followed by the “tab” key.

```
ubuntu$ lsm<tab>
```

9. **Create a second new file named “myhugelongfilename2” by using the terminal history.**

By pressing the up and down arrows, you can cycle through the history of commands that have been executed in the terminal. The first up arrow press will give you the list command of step 8, and the second press will give you the touch command of step 7.

Once the touch command is displayed in the terminal, you can move the cursor within the command using the left and right arrow keys. Use the right arrow key to move to the end of the line (or press the “end” key to move there in one step) and replace the number “1” with “2” and press enter.

**10. Explore the behavior of the autofill when there is a filename conflict.**

```
ubuntu$ ls m<tab>
```

If you were careful to press <tab> just once, you should now see.

```
ubuntu$ ls myhugelongfilename
```

Because both “myhugelongfilename1” and “myhugelongfilename2” meet the criterion “m\*” the autofill has completed as much as it can up to the conflict.

Another useful feature of the autofill is that if you have a conflict, you can press <tab> twice in rapid succession in order to print a listing of all files that meet the current criteria.

```
ubuntu$ ls myhugelongfilename<tab><tab>
```

This should print “myhugelongfilename1” and “myhugelongfilename2” in the terminal. Note that after these files are listed, the prompt returns to:

```
ubuntu$ ls myhugelongfilename
```

This allows you to enter the next character and resolve the conflict. Finish by adding either “1” or “2.”

**11. Edit the “myhugefilename1” file using the “gedit” command.**

```
ubuntu$ gedit myhugefilename1
```

Did you use the autofill (tab) capability? It is a good idea to get into the habit of using the autofill. You might be surprised how helpful it will become as you use Linux more frequently.

**12. Add some text into “myhugefilename1.”**

gedit is a user-friendly text editor that has features that should be familiar to any Windows user, including copy and cut-and-paste with the familiar “ctrl-c,” “ctrl-x,” and “ctrl-v” combinations.

Add anything you like into the file – the name of your first dog, the state where you were born or your favorite flavor of ice cream – and then save the file with the “save” button.



Exit the gedit application by pressing the “x” in the top right corner of the window.

**13. Print the contents of “myhugefilename1” from the terminal with “cat.”**

You can print the contents of a text file using the “cat” command in Linux. (This is short for “concatenate.” If you specify more than one filename to the “cat” command, they will be concatenated together and then displayed on the terminal.)

```
ubuntu$ cat myhugefilename1
```

**14. Print the contents of “myhugefilename1” from the terminal with “more.”**

The “more” command has a number of useful features. If the file you are printing is too large to fit on one terminal screen, “cat” will simply dump everything so that you have to scroll back to see the whole file. “more” will pause after each page, and you may use the space bar to advance to the next page.

```
ubuntu$ more myhugefilename1
```

**15. Create a subdirectory named “subdir.”**

Refer to step 5 if needed

**16. Create a copy of “myhugefilename1” named “mycopy1,” where the file “mycopy1” is in the “subdir” directory.**

```
ubuntu$ cp myhugefilename1 subdir/mycopy1
```

**17. Create a second copy using the current directory reference “.”**

```
ubuntu$ cp ./myhugefilename1 ./subdir/mycopy2
```

The single period “.” in the above command refers to the current directory. It is not necessary to reference the current directory in most instances because it is implied, but there are cases where the current directory reference is useful to know, so we wanted to introduce it here.

**18. Verify the copies by listing the contents of the subdir directory.****19. Change into the subdir directory.****20. Change the name of “mycopy1” to “mynewfilename.”**

```
ubuntu$ mv mycopy1 mynewfilename
```

The Linux “mv” (move) command may also be used to change a filename.

**21. Move “mynewfilename” from the “subdir” directory up one level to the “lab02” directory.**

```
ubuntu$ mv mynewfilename ../mynewfilename
```

Note that the double period (“..”) in the above command refers to the directory one level up from the current directory. Before you ask, there is no triple period. To move up two directories, you would use “../..”

In this case, the same effect could have been achieved with:

```
ubuntu$ mv mynewfilename ..
```

If you move a file to a directory, Linux will automatically retain the original file name.

**22. Copy everything from the “subdir” directory up one level to the “lab02” directory.**

```
ubuntu$ cp * ..
```

The asterisk (\*) acts as a wildcard in Linux just as in DOS and Windows. An asterisk by itself will match everything in the current directory, thus the above command copies everything from the current directory into the parent directory.

**23. Remove “mynewfilename” from the “subdir” directory**

```
ubuntu$ rm mynewfilename
```

**24. Change directories up one level to the “lab02” directory****25. Remove the “subdir” directory and all of its contents (recursive removal)**

```
ubuntu$ rm -R subdir
```

## 2 Linux Redirection

Note that these exercises are to be completed on the Ubuntu host computer and not on the AM335x starter kit.

**26. If you haven't already done so, create "~/lab02" subdirectory and change into it.**

**27. Test the echo command with something like "Linux Rocks!"**

```
ubuntu$ echo Linux Rocks!
```

The echo command takes the string passed to it as a parameter – in this case the phrase "Linux Rocks!" – and writes it to the application's standard output. When you execute a program from a terminal, the standard output is displayed on that terminal. Thus, you see the phrase repeated (echoed) onto the terminal.

**28. Redirect the echo command standard output into the file "test."**

```
ubuntu$ echo Linux Rocks! > test
```

**29. Repeat step 28 with a new phrase, being sure to use a single-greater-than (">") redirection.**

```
ubuntu$ echo AM335x Rocks! > test
```

**30. Dump the contents of the file "test" into the terminal.**

```
ubuntu$ cat test
```

**31. Redirect another phrase into the file "test," this time using double greater-than redirection (">>")**

```
ubuntu$ echo and so does Texas Instruments! >> test
```

**32. Dump the contents of the file "test" into the terminal.**

Have you determined the difference between single-greater-than and double-greater-than redirection? Single greater-than redirection will overwrite the output file, whereas double greater-than redirection will append to it.

**33. Redirect the contents of "test" into the standard input of the command "grep."**

```
ubuntu$ grep Texas < test
```

The grep command will search for a phrase (such as "Texas") in a file or standard input stream and filter out those lines which contain the phrase.

**34. Simultaneous standard input and standard output redirection**

```
ubuntu$ grep Texas < test > test2
```

**35. Dump the test2 file**

You should see that the file test2 has been filled with the same output you previously saw displayed on the terminal in step 33.

**36. Pipe the standard output of the "cat" command into the standard input of the "grep" command.**

```
ubuntu$ cat test | grep Texas
```

The greater-than and less-than signs are referred to as redirection, whereas the "|" is termed a pipe. Piping the output of one command into the input of another is common practice in Linux, especially when writing scripts.



## (Advanced) Linux Scripting

37. If you haven't already done so, create "~/lab02" directory and change into it.

38. Edit a file named "myscript.sh"

```
ubuntu$ gedit myscript.sh
```

39. Create a script that will append "mystring" to any file in the current directory that contains the key phrase "mykey"

The calling format for this script is:

```
ubuntu$ ./myscript.sh mystring mykey
```

A few notes about Linux scripting:

(For more information, refer to "Learning the Bash Shell" by O'Reilly Publishing.)

- a. The first argument passed to the script may be referenced with "\$1" (without the quotes), the second with "\$2", etc.
- b. You can create a for loop from any space-separated list using the following format. The below example script would perform similarly to the "ls" command:

```
for file in ./*
do
    echo ${file}
done
```

- c. You can test with an if statement. The following example script will print the name of the variable "file" only if it matches the string "testfile"

```
if [ "${file}" == "testfile" ]
then
    echo ${file}
fi
```

note that the variable "\${file}" is in quotes. This is for proper handling of the case in which the variable "\${file}" is empty. Without the quotes, there would be a syntax error because there is no left-hand side to the test. With the quotes, the test will hold correctly in the case of an empty variable because the left-hand side is an empty string.

Note that the space after the initial "[" bracket is required, as is the space before the final bracket.

- d. You can test if a file contains a given string with the "grep" command. grep is a very useful search utility, which along with "sed" (a serial editor utility which can do search-and-replace) is very commonly used in Linux scripting. To test if the file "myfile" contains the string "mykey," you could use the test:

```
if [ `grep -c mykey myfile` -ne 0 ]
```

Passing the "-c" option will instruct grep to count the occurrences of the following string. By placing the full command in right-facing single quotes, the command is executed in the shell and the result is inserted in place of the expression. Note that both the initial and final quotation marks must be right-facing.

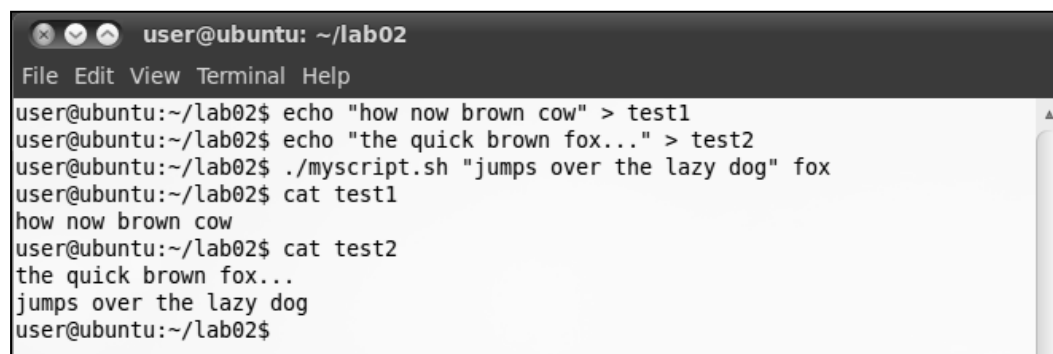
40. Change the permissions of “myscript.sh” to add executable permission for all users

```
ubuntu$ chmod a+x myscript.sh
```

41. Create one or more test files in the “lab02” directory and test your script.

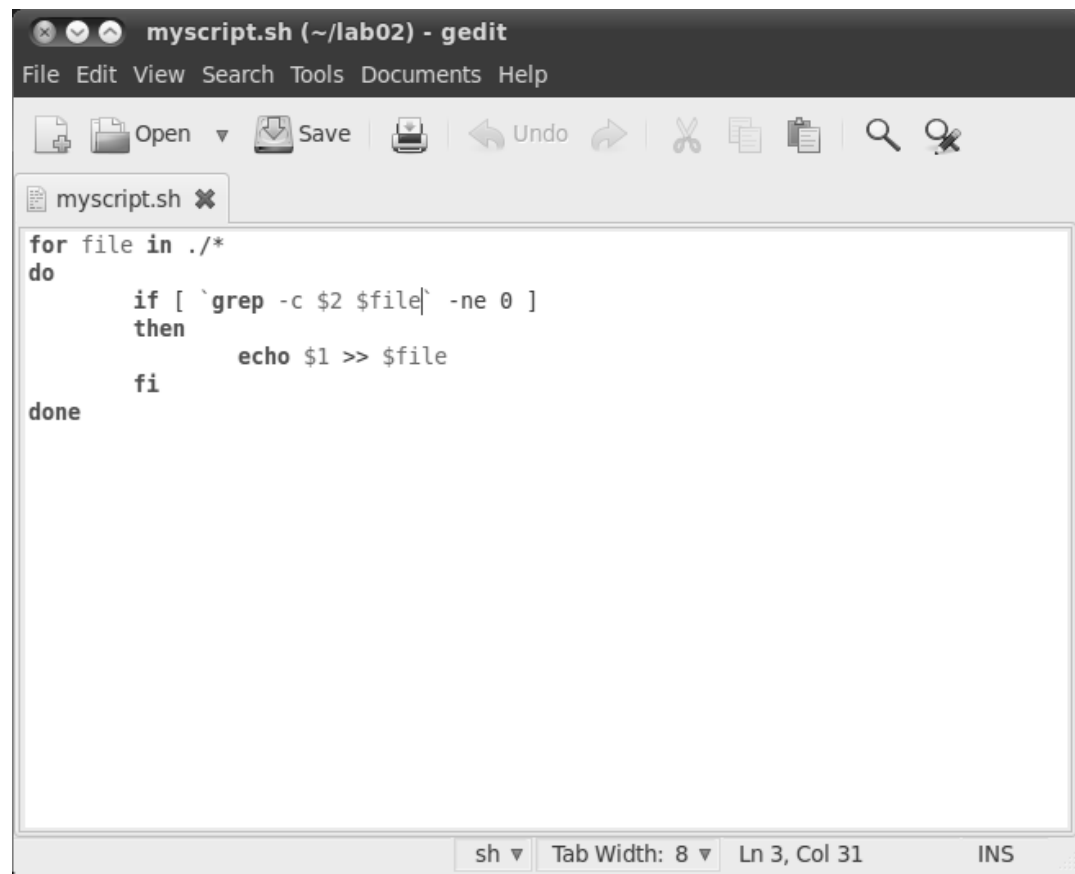
**Note:** The solution is shown on the next page.

Correct operation is as shown below:

A terminal window titled 'user@ubuntu: ~/lab02' with a menu bar (File, Edit, View, Terminal, Help). The terminal shows the following commands and output:

```
user@ubuntu:~/lab02$ echo "how now brown cow" > test1
user@ubuntu:~/lab02$ echo "the quick brown fox..." > test2
user@ubuntu:~/lab02$ ./myscript.sh "jumps over the lazy dog" fox
user@ubuntu:~/lab02$ cat test1
how now brown cow
user@ubuntu:~/lab02$ cat test2
the quick brown fox...
jumps over the lazy dog
user@ubuntu:~/lab02$
```

**42. Solution:**



```
mysript.sh (~/.lab02) - gedit
File Edit View Search Tools Documents Help

mysript.sh x
for file in ./*
do
    if [ `grep -c $2 $file` -ne 0 ]
    then
        echo $1 >> $file
    fi
done

sh ▾ Tab Width: 8 ▾ Ln 3, Col 31 INS
```

**(Page intentionally left blank)**

# Lab 3: OpenEmbedded

---

## Introduction

In this lab exercise you will explore OpenEmbedded, which is a Bitbake-based system for building entire distributions of Linux from recipe source files.

You will begin by investigating the workshop Bitbake layer that has been added ontop of the OpenEmbedded Arago distribution recipes.

In the second section you will then create a build recipe for the provided “Hello World” C source code.

## Module Topics

<b>Lab 3: OpenEmbedded .....</b>	<b>3-1</b>
<i>Module Topics.....</i>	<i>3-2</i>
<i>1 Exploring the Workshop Bitbake Layer .....</i>	<i>3-3</i>
<i>2 Write a Basic C-code Recipe .....</i>	<i>3-6</i>

# 1 Exploring the Workshop Bitbake Layer

One of the fundamental tenets of the Bitbake tool is that, when modifying an existing distribution, you should make changes in the form of an overlay layer instead of modifying the base recipes of the existing distribution. This is equivalent to the concept of inheritance that is used throughout object-oriented programming.

The workshop exercises are implemented in Bitbake recipes which are collected into a Bitbake layer called “meta-workshop.” By convention, the names of Bitbake layers begin with “meta-,” although this is not a requirement.

## 1. Locate the Yocto/OpenEmbedded source recipes

```
ubuntu$ cd /home/user/oe-layerssetup/sources
ubuntu$ ls
```

Here you see the various layers that comprise the recipe system for the OpenEmbedded build system. The layers are organized as follows:

bitbake                      The fundamental recipes and class definitions of the Bitbake build system. The recipes and class definitions provided here are the most fundamental, and nearly every type used in an OpenEmbedded build inherits from a definition provided here. Note that OpenEmbedded, while utilizing the Bitbake tool, is still a separate entity and has its own recipes as defined in “oe-core.”

meta-arago                  This layer extends the OpenEmbedded base recipes found in the “meta-openembedded” layer in order to define the Arago Linux distribution.

meta-linaro                This layer contains recipes for building the Linaro toolchain. Linaro provides an open-source compiler, assembler, linker, etc. Currently the Linaro toolchain is recommended for cross-compiling the Arago distribution for use on TI’s Sitara devices.

meta-openembedded        This layer contains a large array of basic recipes for building various common components of a Linux distribution from Gnu-community source code.

meta-qt5                    This layer contains recipes for building qt version 5. Currently only qt version 4 recipes have been integrated into the “meta-openembedded” layer and this latest version of qt is maintained in a separate layer.

meta-ti                     This layer contains recipes and class definitions specific to Texas Instruments. The dividing line between “meta-ti” and “meta-arago” is somewhat hazy, but generally this layer contains more hardware-specific information such as patches for taking advantage of various hardware accelerators on TI devices.

meta-workshop            This layer contains recipes for the lab exercises of the Texas Instruments “Introduction to Embedded Linux” workshop series.

oe-core                    The fundamental definitions of the OpenEmbedded community. Whereas bitbake provides a generalized build system, these definitions provide build mechanisms specifically tailored towards building a Linux distribution such as kernel and bootloader build recipes and recipes for adding initialization scripts.

## 2. Display the Arago Bitbake layers

```
ubuntu$ cd /home/user/oe-layerssetup/build
ubuntu$ source conf/setenv
ubuntu$ MACHINE=am335x-evm bitbake-layers help
```

The output of this command shows various commands accepted by the “bitbake-layers” utility.

```
ubuntu$ MACHINE=am335x-evm bitbake-layers show-layers
```

Note that some of the subdirectories listed in step 1 actually contain multiple layers. This utility shows each layer, its location and its priority. Layers of a higher priority will override those of a lower priority.

You can see that the “meta-workshop” layer is priority 11 and the various Arago layers are all priority 10 or less. This means that the “meta-workshop” layer will override the definitions of the base Arago distribution.

## 3. Explore the meta-workshop layer

```
ubuntu$ cd /home/user/oe-layerssetup/sources/meta-workshop
```

## 4. View the layer configuration file

Each bitbake layer must contain a layer definition and configuration file at <layer-path>/conf/layer.conf

```
ubuntu$ more conf/layer.conf
```

This file modifies the following environment variables used by the Bitbake tool:

**BBPATH** This environment variable is used to search for conf and class files. The search directories form a colon-separated list. By appending the path of this layer to the variable, this layer’s “conf” and “classes” directories will be searched.

**BBFILES** This environment variable provides search directories for various bitbake recipe files within the layer, i.e. those files with “.bb” and “.bbappend” extensions. Note the use of the wildcard character “\*” to specify every one of these files in the layer. This variable is a space-separated list. Note that if subdirectories are added to the layer, they must be added to this variable, or else the bitbake tool will not locate them.

**BBFILE\_COLLECTIONS** This variable allows advanced users to separate the layer into various collections (see below). Most users will simply define a single collection which has the same name as the layer.

**BBFILE\_PATTERN\_<collection-name>** This variable is set with a regular expression that can be used to filter the recipes of the layer into various collections. Note that this pattern is only applied to those files which are specified in “BBFILES.” Generally there is a single collection defined for each layer, in which case the regular expression “^\${LAYERDIR}/” will include every file in the layer and does not need to be modified.

**BBFILE\_PRIORITY\_<collection-name>** This environment variable sets the priority of the “meta-workshop” collection. Note that this value in this file determines this priority and is therefore the same as was displayed in step 2.



**5. List the contents of the meta-workshop layer**

```
ubuntu$ ls
```

You will see the following subdirectories:

conf                      directory for configuration files specific to the layer

images                    recipes that define distribution images (i.e. all the files one would copy to an sd card to boot a Linux distribution) and SDK images. The files defined here integrate the recipes in the other subdirectories into the “arago-core-tisdk-image” recipe, allowing a specialized workshop version of the SDK to be built.

packagegroups            recipes that group individual Bitbake recipes together into package groups. These recipes utilize the “packagegroup” class.

recipes-overlay            recipes that overlay (i.e. override) the behavior of one of the underlying Arago distribution layers.

recipes-workshop-c        this subdirectory contains recipes for rebuilding the workshop lab files which are written in the form of C or C++ code.

recipes-workshop-qt      this subdirectory contains recipes for rebuilding the workshop lab files which are written in the form of qt projects.

## 2 Write a Basic C-code Recipe

In this section, you will write a basic Bitbake recipe to build a “Hello World” application that has been provided for you in C code.

### 6. Change to the “lab03\_student” directory

Assuming that you are already in the “/home/user/oe-layersetup/sources/meta-workshop” directory, which should be the case upon completing section 1,

```
ubuntu$ cd recipes-workshop-c/lab03_student
ubuntu$ ls -R
```

You will find the following files and directories:

files/hello.c	“Hello World” application
files/LICENSE	(empty) license file
lab03-student-native_1.0.bb	An incomplete recipe file for executing a native build. Recall that a native build is one that will execute on the computer that is used to build the recipe (as opposed to the default behavior which would be to build for a specific target such as the AM335x Starter Kit.) This will allow you to test your application on the x86 Ubuntu machine you are using to build it.

### 7. Determine an md5 sum (hash) for the license file

OpenEmbedded now forces recipes to have an associated LICENSE and LICENSE file specifying the terms. Recipes are required to specify the md5 sum of the LICENSE file in order to avoid either accidental or malicious tampering of the file.

```
ubuntu$ md5sum files/LICENSE
```

### 8. Edit the Bitbake recipe

```
ubuntu$ gedit lab03-student-native_1.0.bb
```

Note the line:

```
LIC_FILES_CHECKSUM =
"file://LICENSE;md5=d41d8cd98f00b204e9800998ecf8427e"
```

This specifies a license file for the recipe and the md5 sum for the license file. It should match what you determined in step 7.

### 9. Declare native class inheritance

Add the following declaration to lab03-student-native\_1.0.bb:

```
inherit native
```

The inherit declaration will override the base class variables and build tasks with tasks from the “native” class. These task overrides will ensure the the application is built for the i686 (native processor) instead of the Cortex-A8.

## 10. Enter source Uniform Resource Identifier (URI)

The SRC\_URI variable is a space-separated list of URI-formatted files. A URI is similar to a URL (Uniform Resource Locator) if you are familiar with that terminology. URI's are more generic which is to say that every URL is a URI, but the converse is not necessarily true.

Since all of the files that we are using for source code are locally resident, the URI will use the "file://" prepended identifier. If you were downloading from the internet you might use "ftp://," "http://," "https://," "git://," etc.

You will not need to specify an absolute path to the files you are including as source files. In fact, not only do you not need to use absolute paths, but you should avoid them in order to make your recipe relocatable. Bitbake will automatically search the "files" directory within the recipe, so you only need to specify the path from this point, i.e.:

```
"file://LICENSE"
```

In addition to the LICENSE file, you will need to include "helloworld.c" Recall that the SRC\_URI variable is a space-separated list of URI values.

If you wish, you may span the definition of the SRC\_URI variable across multiple lines by terminating one line with the backslash (\) character and then entering a carriage return. The carriage return will be ignored (because it is escaped with the backslash character directly preceding it) but will appear when the text-based recipe file is displayed in an editor.

You may either specify "helloworld.c" as a direct value or utilize the wildcard (\*) character in order to make your definition more generic using "\*.c"

## 11. Set the source directory (\${S}) to match the working directory (\${WORKDIR})

The default "do\_fetch" task will copy all files specified in the "SRC\_URI" variable into the working directory, \${WORKDIR}. The default "do\_unpack" task will de-archive any recognized archive files (.tar.gz, .tar.bz2, etc) into the source directory, \${S}.

Our source files, however, are not in the form of an archive, so the "do\_unpack" task will do nothing, leaving the source files in the \${WORKDIR} directory. There are two solutions. Perhaps the more elegant solution would be to override the "do\_unpack" task to copy our source files from \${WORKDIR} into \${S}. Instead, we choose the simpler solution of setting \${S} equal to the \${WORKDIR}.

```
S="${WORKDIR}"
```

Note that within the bitbake context, the quotation marks around the \${WORKDIR} variable expansion are required for proper operation.

## 12. Enter `do_compile` function

The default implementation of the “`do_compile`” function is to call “`make`.” This is fine if your source code includes a makefile, but ours does not! One solution would be to create a makefile, but instead lets just redefine the `do_compile` function to meet our needs. Begin by removing the colon (:) character from the “`do_compile`” function. This symbol allows the definition of an empty function without generating a python error.

Next fill in the `do_compile` function. The function uses BASH shell syntax. You should be able to implement the compile step in a single line. A few pre-defined variables that will be useful for you are:

<code>\${CC}</code>	The gnu C compiler chain, including path on this system
<code>\${S}</code>	The source code directory
<code>\${CFLAGS}</code>	A set of C compiler flags that are appropriate for this target
<code>\${LDFLAGS}</code>	A set of linker flags that are appropriate for this target

Be sure to name the output file “`lab03-student.`” (A variable `${MYBASENAME}` has been created for this purpose.)

It should be noted that even though you could implement “`do_compile`” without using the above variables, you should use these variables to make your recipe portable across multiple targets.

## 13. Enter `do_install` function

The default “`do_install`” function is to call “`make install.`” Since we do not have a makefile, we will want to override this function with our own definition.

As before, remove the colon (:) character from the empty “`do_install`” function. You will then need to add two lines for this function. The first line will create an empty directory to store the binary built by “`do_compile.`” The second line will actually copy the binary executable generated by “`do_compile`” into this directory.

A few pre-defined variables that will be useful to you:

<code>\${D}</code>	The destination directory, which warehouses the package files
<code>\${bindir}</code>	The path on the target to the directory holding this package’s binaries

## 14. Save file

File→Save (ctrl-s)

## 15. Build recipe

```
ubuntu$ cd /home/user/oe-layerssetup/build
ubuntu$ MACHINE=am335x-evm bitbake lab03-student-native
```

## 16. Test application

Enter the following (all one line without carriage returns)

Hint: Don’t forget the autofill (tab) feature!

```
ubuntu$ ./arago-tmp-external-linaro-toolchain/
sysroots/i686-linux/home/user/workshop/
lab03_workspace/lab03-student/lab03-student
```

If successful, you will see “Hello World” displayed on the terminal.

**17. Compare your method to the solution**

There are more than one correct means of writing this recipe. Compare yours to the solution and make note of any differences.

```
ubuntu# more ../lab03-solution-a/  
lab03-solution-a-native_1.0.bb
```

(Page intentionally left blank)

# Lab 4: Code Composer Studio Debug

---

This lab exercise explores using CCSv6 (i.e. Eclipse) for building and debugging our Linux applications. First, you will import your OpenEmbedded recipe as a Makefile project and rebuild the application. Next, you will configure remote debugging and then finally run/debug the program.

In the case of Linux applications, it's often convenient to use the GDB (Gnu DeBugger) protocol – running over Ethernet (TCP/IP) – for connecting between the host (CCSv6/Eclipse) and the target (Linux application running on the ARM processor).

## CCSv6 Installation

We've installed CCSv6 into the `/home/user/ti` folder

## Module Topics

<b>Lab 4: Code Composer Studio Debug .....</b>	<b>4-1</b>
<i>Module Topics.....</i>	<i>4-2</i>
<i>Test Secure Shell Protocol .....</i>	<i>4-3</i>
<i>Create CCS Project.....</i>	<i>4-5</i>
<i>Setup CCSv5 Remote System Explorer Connection .....</i>	<i>4-11</i>
<i>Setup CCSv5 Debug Configuration .....</i>	<i>4-18</i>



# Test Secure Shell Protocol

## 1. Boot the AM335x starter kit and log in as “root” if you have not already done so

You need to either have an Ethernet cable connected between the AM335x starter kit and the host computer, or you need a USB cable with the Ethernet-over-USB gadget driver. The lab instructions will assume you are using standard Ethernet, but may be easily modified to use Ethernet-over-usb by changing the “192.168.1.x” subnet address to the “192.168.7.x” subnet address.

Refer to “Lab01: Booting Linux” for more information.

## 2. Verify IP connection

```
ubuntu$ ifconfig
```

## 3. Verify static IP settings

The IP addresses for the eth0 and usb0 connections have been statically set in a system configuration file at “/etc/network/interfaces.” You can see the details using:

```
ubuntu$ more /etc/network/interfaces
```

Scroll through the file until you locate the entry for the “eth0” interface:

```
auto eth0

iface eth0 inet static
    address 192.168.1.1
    netmask 255.255.255.0
```

There is a similar entry for the usb0 interface.

## 4. View hostname definitions

```
ubuntu$ more /etc/hosts
```

Linux provides a few mechanisms for resolving hostnames. The more complex method uses the Domain Name System (DNS) to resolve hostnames from databases stored on special servers named “Domain Name Servers.”

A simpler mechanism allows system administrators to store hostnames in a system file at “/etc/hosts.” This method only allows for the specification of static IP addresses, and is local to the machine on which the “/etc/hosts” file is located. However, even with these limitations, the simpler method is sufficient for our needs in the workshop.

## 5. Test connection

```
ubuntu$ ping am335x.gigether.net
```

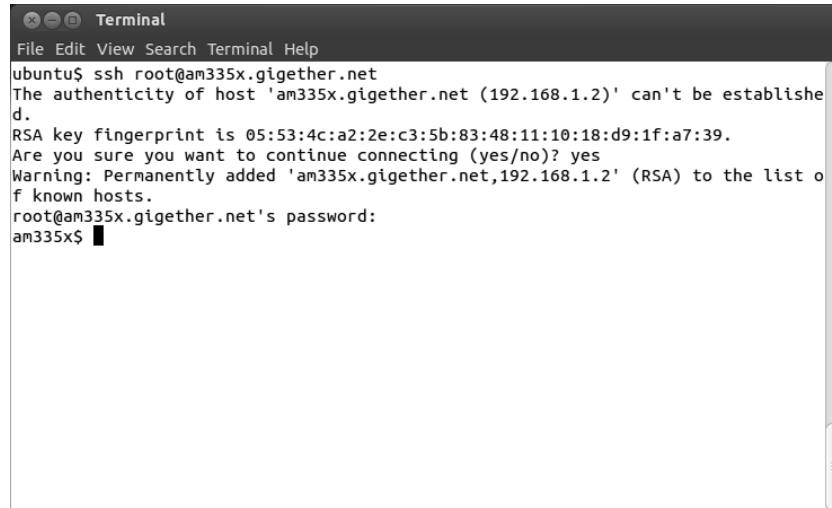
## 6. Establish a secure shell (ssh) connection between host and am335x starter kit

```
ubuntu$ ssh root@am335x.gigether.net
```

Currently there should be no key on file for the hostname you are connecting to and you should see a message:

```
"The authenticity of 'host am335x.gigether.net
(192.168.1.2)' can't be established..."
```

This message is shown in the screen capture below:



If you get this message, type “yes” at the prompt and ssh will generate a new key for you.

## 7. Press “Enter” key when prompted for password

There is no root password on the AM335x starter kit, so when ssh prompts you for a password, simply press “Enter.”

## 8. Explore Terminal

You can navigate an ssh terminal exactly as you would a minicom or screen terminal as you have been using previously.

## 9. Exit from ssh

```
am335x$ exit
```

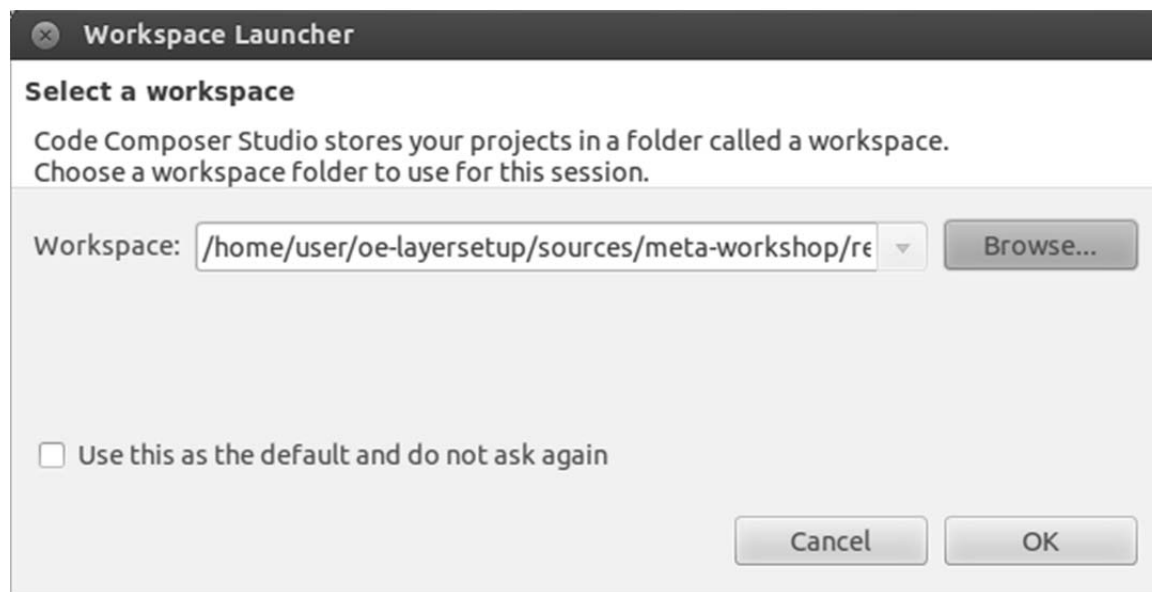
(From within the terminal where you launched ssh)

## Create CCS Project

1. Start CCSv6 from the Ubuntu desktop.
2. Select “/home/user/oe-layersetup/sources/meta-workshop/recipes-workshop-c/lab04\_workspace” as your workspace

Note: Use the “Browse...” button!

By default, CCS will query you for the workspace you would like to use each time it starts as per the following window:



The workspace contains general information that is outside the scope of individual projects such as the debugging connections that have been configured and the general settings for the IDE. Each lab exercise is organized into its own workspace for this workshop.

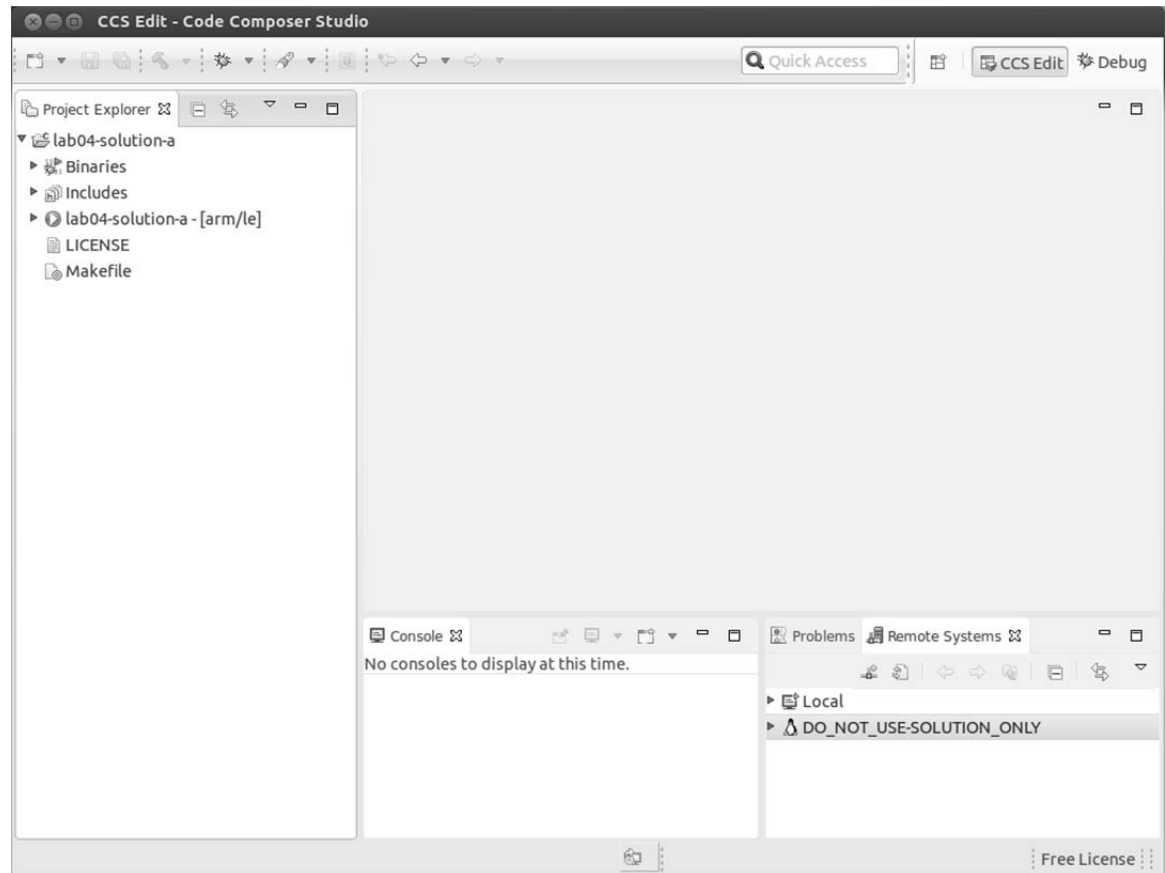
If you choose to select “Use this as the default and do not ask again” on this screen, you can switch between workspaces after CCS has loaded by using

File→Switch Workspace

### 3. Examine the lab04\_workspace

Currently there is one project in the workspace, “lab04-solution-a”

This project was added to the workspace using the same procedure you will now follow for the “lab04-student” project.



The “lab04-solution-a” project contains the “LICENSE” and “Makefile” from the “files” directory of lab04-solution-a. Note that the source code for the project is not shown. This is because the source code for the workshop solutions is maintained on a git repository at [www.github.com](http://www.github.com) and downloaded by Bitbake according to the lab recipe files. As such, this source code is not available to modify within the CCS editor.

If you launch the “lab04-solution-a” debugger, the source code will appear in the debugger.

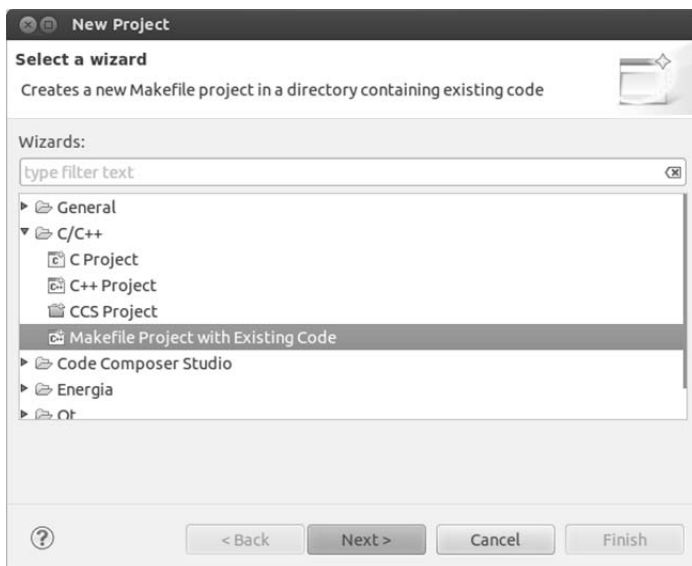
#### 4. Create a new project in the lab04\_workspace workspace



Eclipse provides many different types of projects. The project type selected will affect which build tools are used to build your project

- **CCS Project** – uses Eclipse’s managed make capability, which builds and maintains the make file for you as you add/subtract items and settings from the GUI. By default, a CCS Project builds with the compiler, assembler and linker that are shipped with Code Composer Studio.
- **C/C++→C++ Project** (or C Project) – uses Eclipse’s managed make capability, but gives you a wider range of options for specifying the build tools you wish to use (whereas CCS Project will automatically use the build tools that come with CCS.) We will use this project type and specify the arago distribution gnu compiler chain (gcc) build tools.
- • **C/C++→Makefile Project with Existing Code** – uses your own makefile; while this leaves the work of building and maintaining your own makefiles, it gives you absolute control over your builds
- **Qt** – allows you to import a qtopia project, i.e. a project you created using QT Creator and/or QT Designer.

We’ll create a “C/C++ → Makefile Project with Existing Code” as shown in the following screen capture. Select in the tree and then press “Next.”



CCS will then query the location of the existing Makefile. Browse to “/home/user/oe-layersetup/sources/meta-workshop/recipes-workshop-c/lab04-workspace/lab04-student/files”

Recall that this Makefile has been configured to build the application using the OpenEmbedded build system.

**New Project**

**Import Existing Code**  
Create a new Makefile project from existing code in that same directory

Project Name  
lab04-student

Existing Code Location  
/home/user/oe-layersetup/sources/meta-workshop/recipes-workshop-c/lab04-workspace/lab04-student/files Browse...

Languages  
☒ C ☒ C++

Toolchain for Indexer Settings  
<none>  
Cross GCC  
Linux GCC  
TI Build Tools

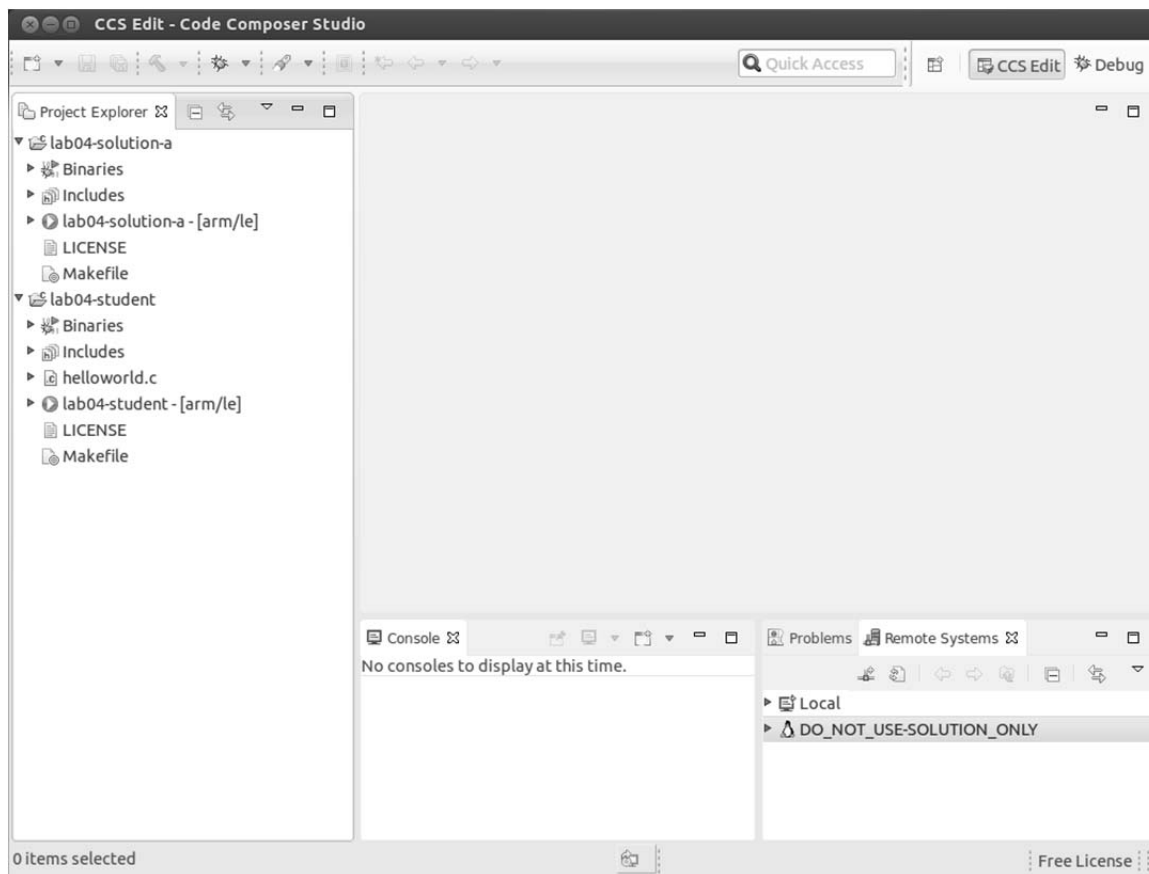
☒ Show only available toolchains that support this platform

? < Back Next > Cancel Finish

The “Project Name” will default to “files” because this is the directory where the Makefile is located. You will want to change to a more descriptive name, such as “lab04-student.”

Also, be sure to select “Cross GCC” as the Toolchain for the Indexer (this will allow CCS to index your source files, enabling autocomplete and other useful features.)

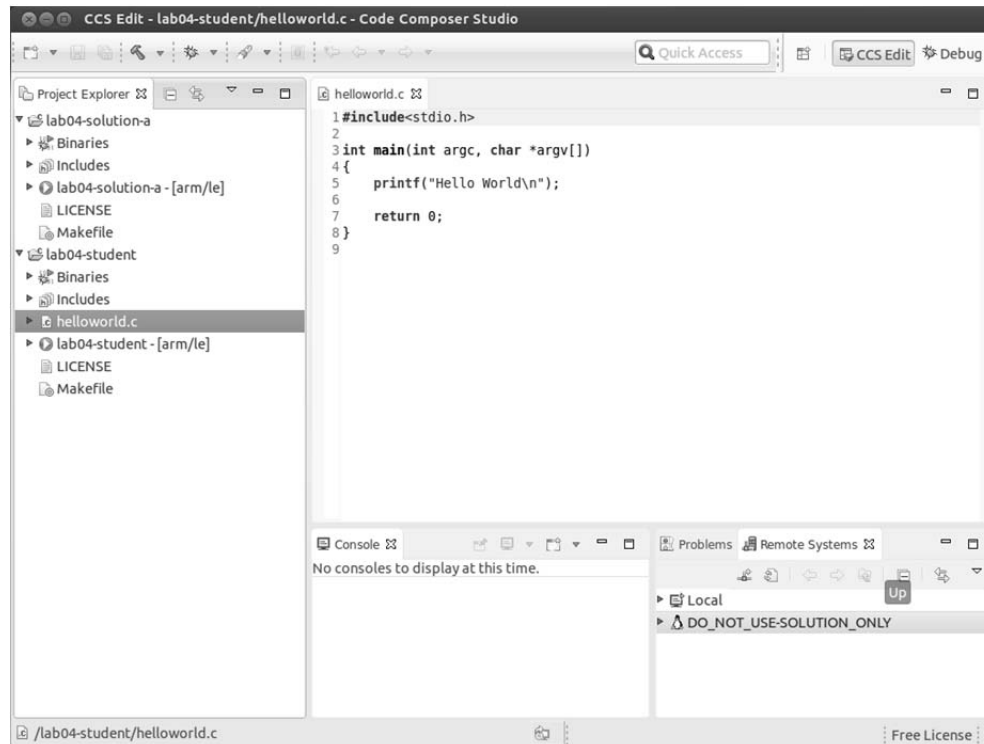
5. Upon successful completion, you should see the “lab04-student” project in the Project Explorer window



Note that the “lab04-student” project displays the “helloworld.c” source code in the project explorer window. This is because this source file is located in the “files” folder of the Makefile that was imported, as opposed to the solution files, which download their source from a git repository at github.

## 6. Examine the provided helloworld.c

You can double-click on the “helloworld” entry in the Project Explorer window



## 7. Build your program

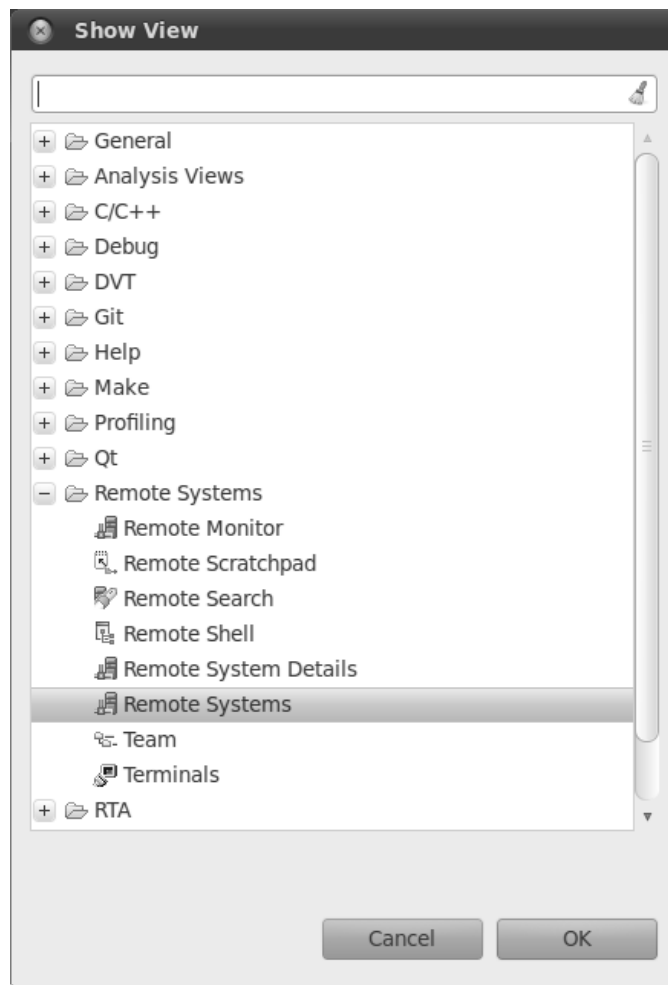
Project→Build All (ctrl-b)



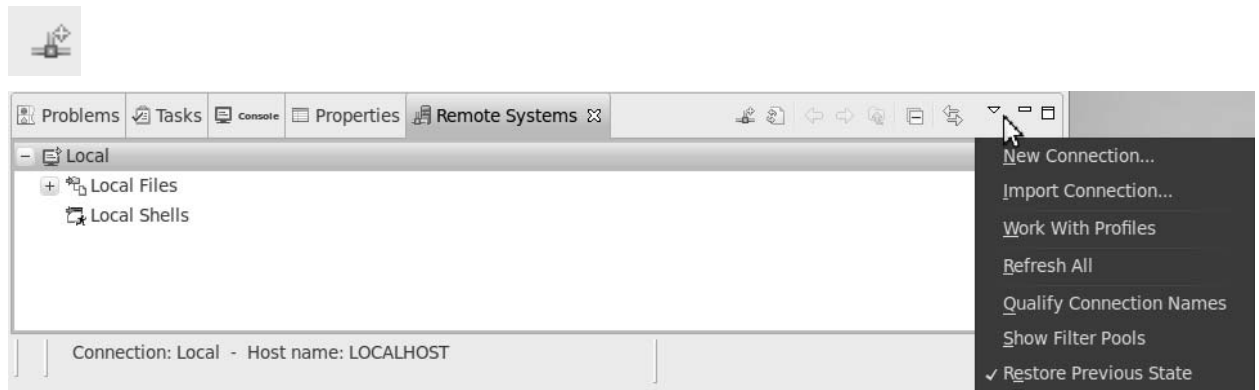
## Setup CCSv5 Remote System Explorer Connection

### 8. Open the remote systems view

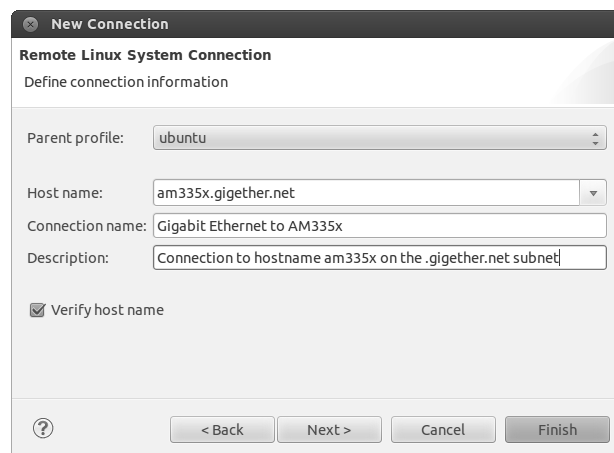
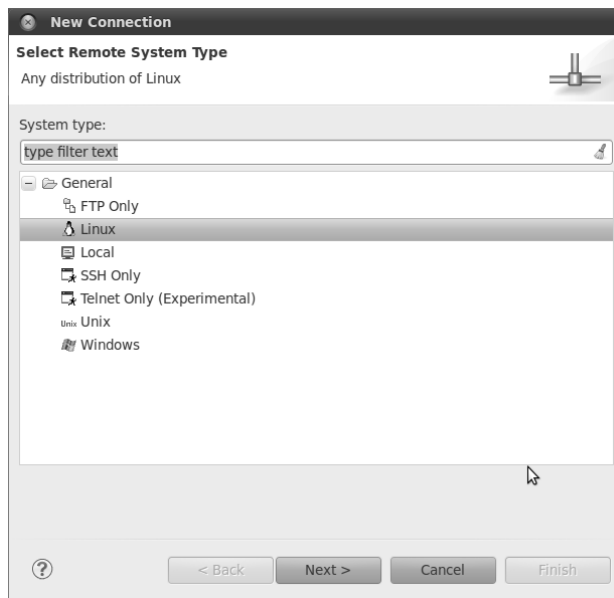
Window → Show View → Other... → Remote Systems → Remote Systems



**9. Configure a new connection using the Remote Systems menu or by pressing the new connection button**



**10. Configure the connection as per next six screen-captures**

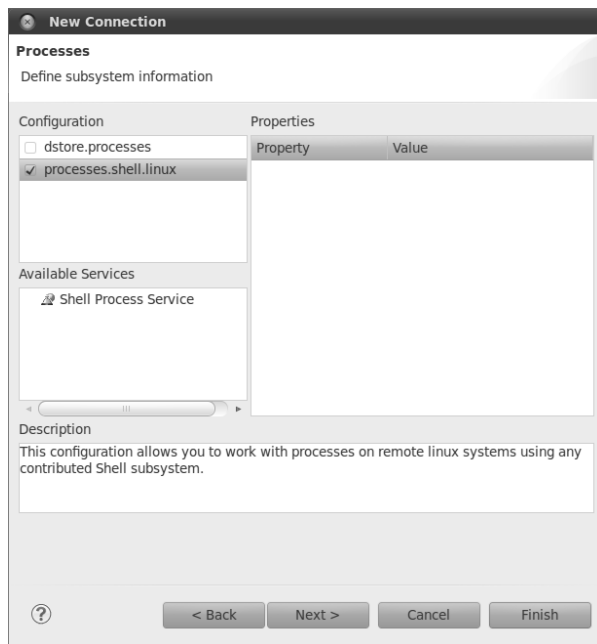
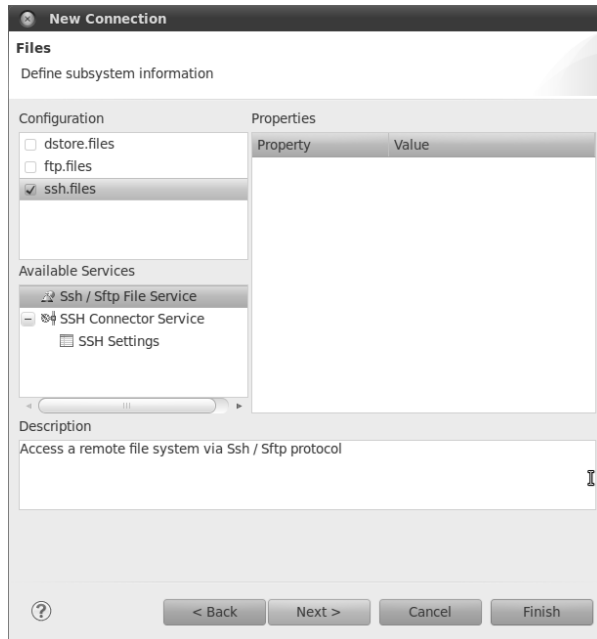


**Remote System Type:** Linux

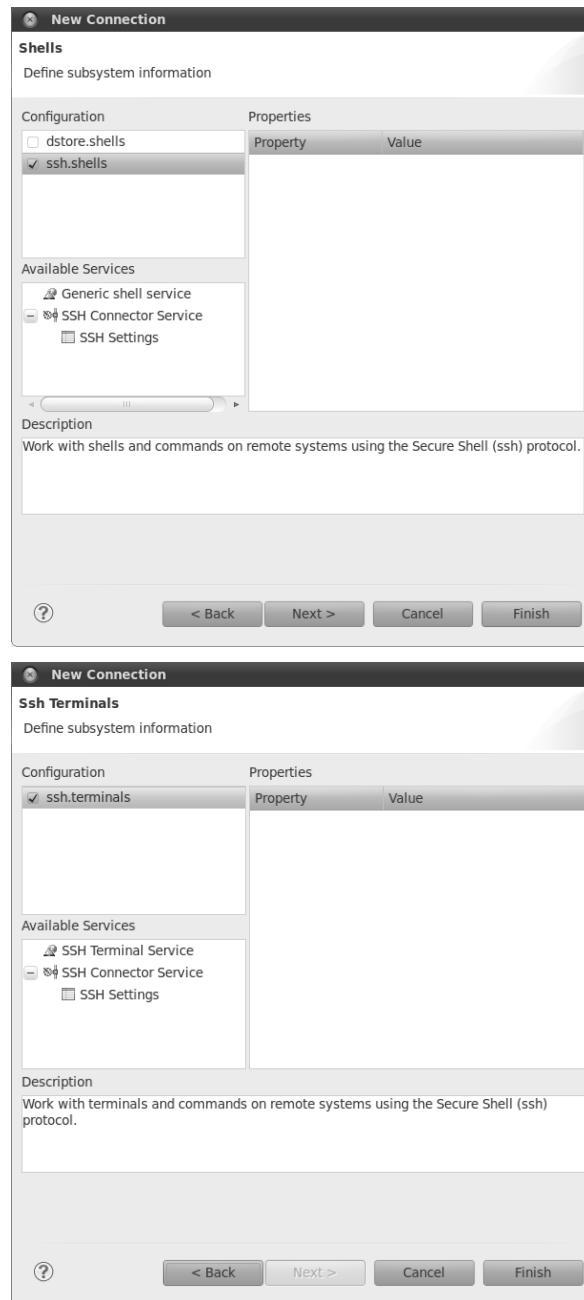
**Host Name:** am335x.gigether.net

**Connection name:** Gigabit Ethernet to AM335x

**Description:** Connection to hostname am335x on the .gigether.net subnet



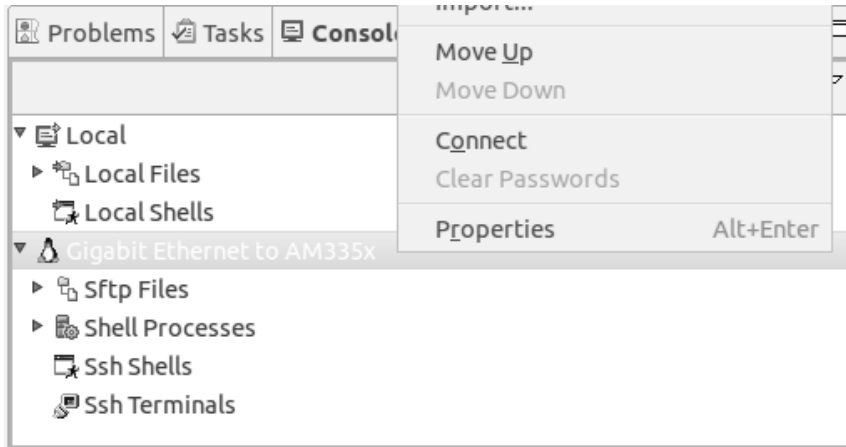
On third screen (files) select “ssh.files” under configuration and on the fourth screen (processes), select “processes.shell.Linux” under configuration.



On the fifth screen (shells) choose “ssh.shells” and on the sixth screen (ssh Terminals) choose “ssh.terminals.” You may press “Finish” on the sixth screen.

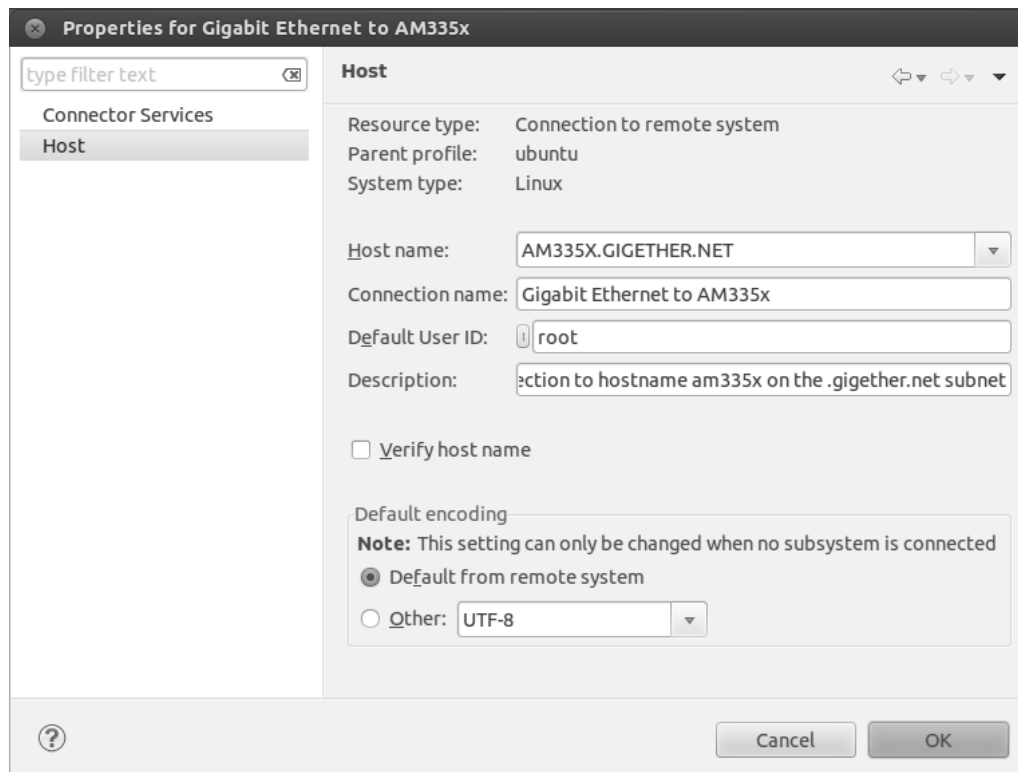
## 11. Change the ssh login user ID to “root”

Right-click “Gigabit Ethernet to am335x” in the Remote Systems window and select “Properties”



Change Default User ID to root as shown:

Note: You may need to press the small button that appears between “Default User ID:” and the entry box if the entry box is grayed over.



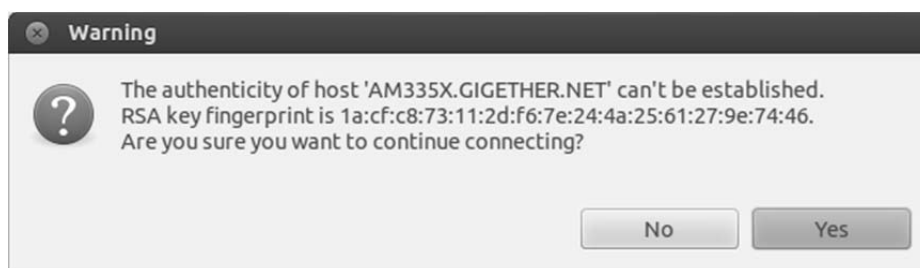
Press OK

## 12. Connect to target using newly created connection

Right-click on “Gigabit Ethernet to AM335x” in the Remote Systems window and select “Connect”

You may get a message indicating the ssh service does not recognize the RSA key fingerprint of the AM335x starter kit. This is the same message that you should have seen in step 6 from the terminal, and it will only appear if you did not generate a new key in that step.

If it appears, press “Yes” to continue connecting and you should not see it a second time.



You may also receive an unknown host error if the “known\_hosts” file was not updated in step 6. If so, press “yes” to continue.

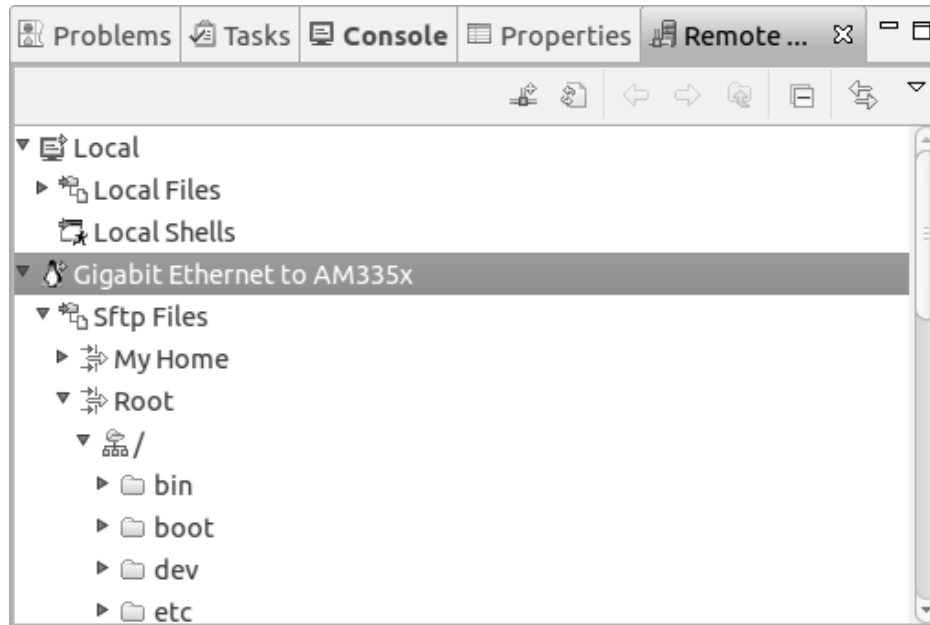


## 13. If queried, enter username and password as follows:



**14. If queried for a Secure Storage password, cancel out fo the query box,****15. Test connection with filesystem viewer**

If you have successfully connected the RSE, you should be able to expand the SFTP files tab under the connection as shown:



## Setup CCSv5 Debug Configuration

### 16. Create a new C/C++ debug configuration.

Bring up the Debug Configurations dialog:

Run → Debug Configurations

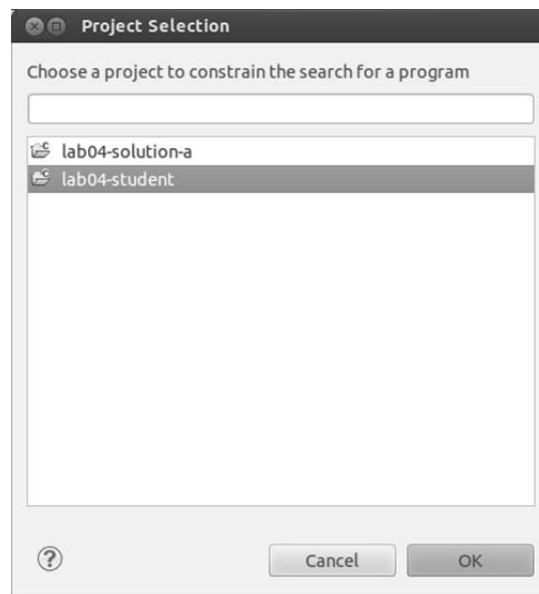
Select "C/C++ Remote Application" and create a new configuration with the icon in the upper left hand corner of the window or by right-clicking "C/C++ Remote Application."

### 17. Setup the configuration:

Use the following settings (many of these should be set by default):

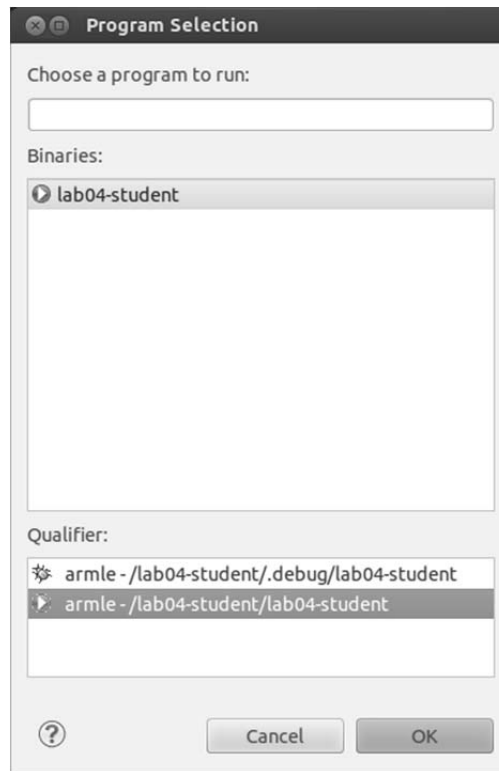
Name: **lab04-student**

Project: **lab04-student** (select using "Browse...")





C/C++ App: **lab04-student** (select using “Browse...”)



Note: It would be intuitive to use the “.debug” version of “lab04-student” as shown on this screen capture, but this will not work. The reason is because the file stored in “.debug/lab04-student” contains only the debugging symbols, but not the executable code.

When gcc loads an executable file, it automatically looks for debug symbols in the “.debug” directory of the same folder, so the correct way to load the executable with debug symbols in this case is to load the stripped executable, so long as the debug symbols are available in the .debug folder.

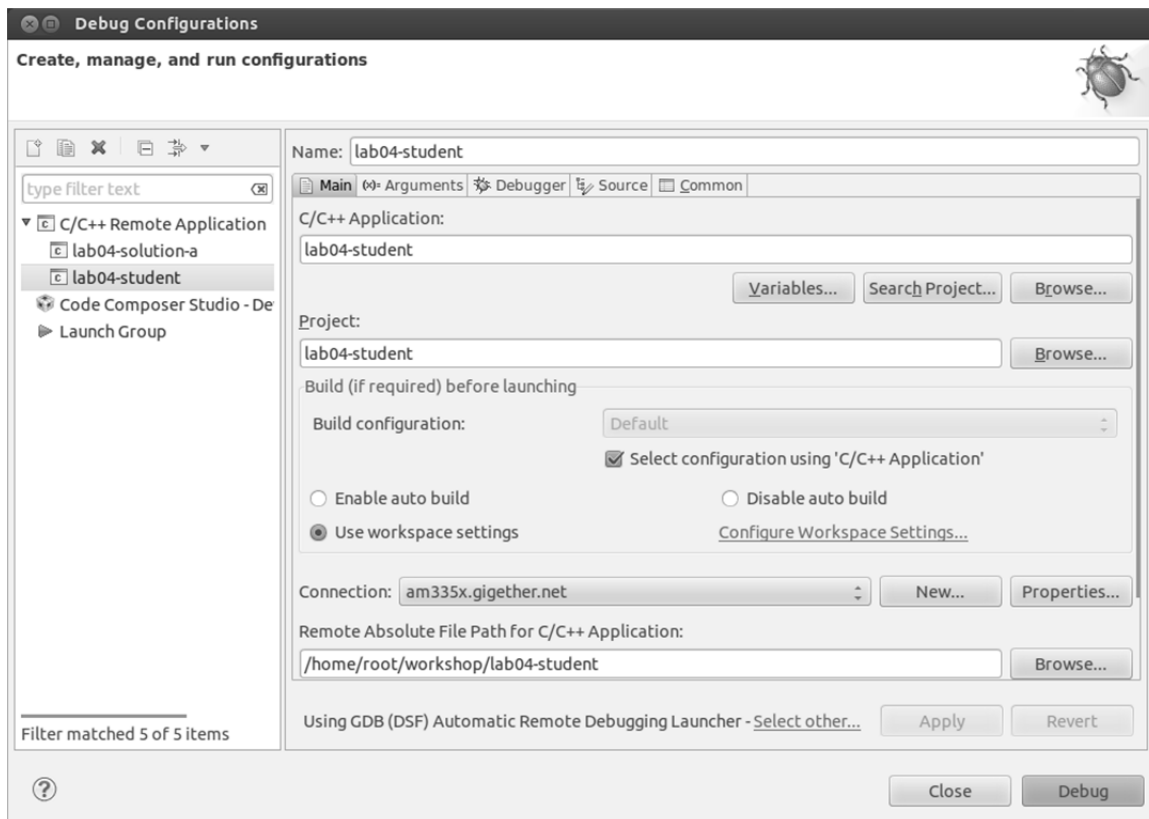
Providing a stripped executable with a symbols-only file in the .debug directory is the default build behavior of OpenEmbedded.

Remote Path: **/home/root/workshop/lab04-student**

The remote path is the location on the remote system where CCS will download the executable before launching it within the gdbserver context.

When finished, this tab should look as follows:

Note: Don't press the "Debug" button yet, we still need to fill out the "Debugger" tab!

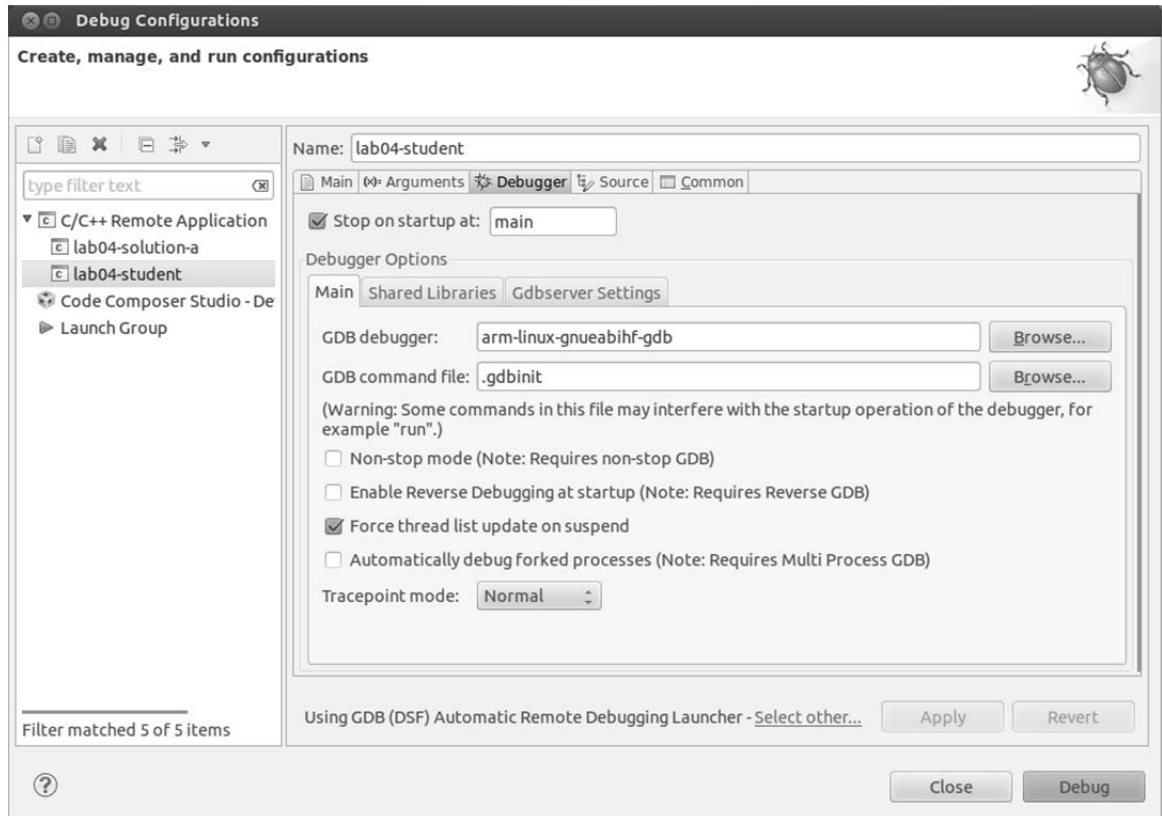


**18. On the *Debugger Main* tab, specify the GDB debugger.**

We are using the cross version of the GDB debugger. The cross version runs on the i686 platform but debugs code for the AM335x device. Thus, we need to specify the GDB debugger as:

```
arm-linux-gnueabi-hf-gdb
```

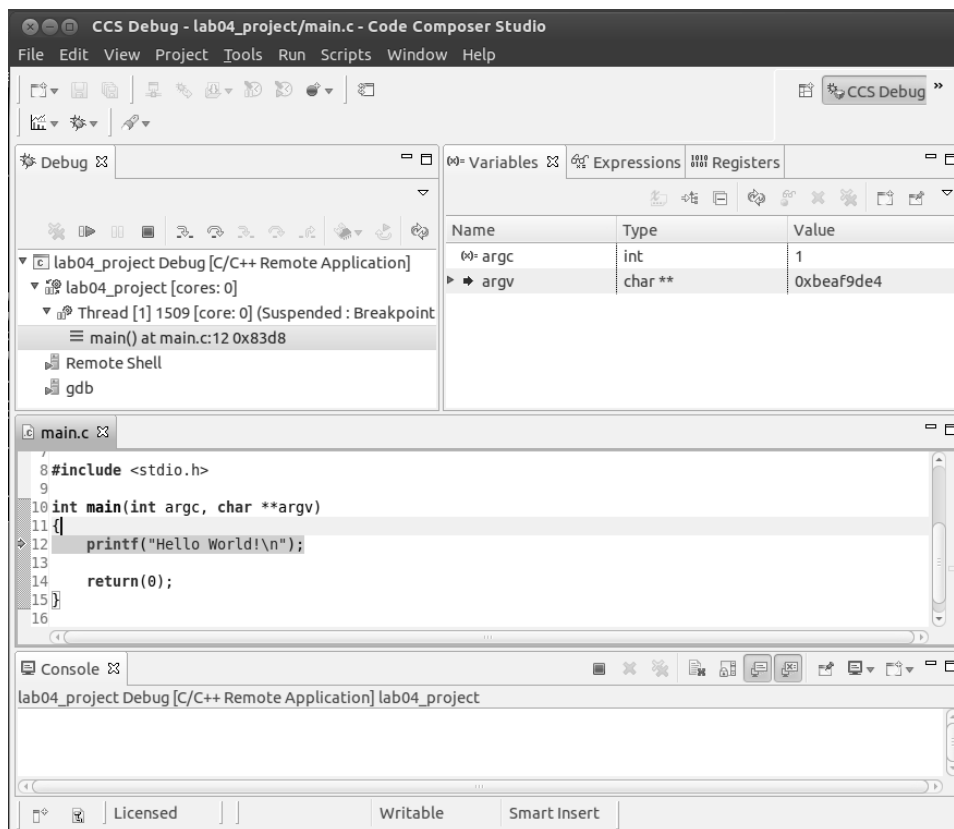
Note that this application must be available in the Linux “PATH” environment variable. (This has been taken care of in the lab setup.)



Note as well that the “Force thread list update on suspend” box has been checked. This is not necessary for this hello world application because it is single-threaded, but it is recommended that you always check this box as multi-threaded applications (see Module 07 and Lab 07) will not debug properly without this box checked. There is also very little overhead to updating the thread list on each suspend, so it might as well be selected each time.

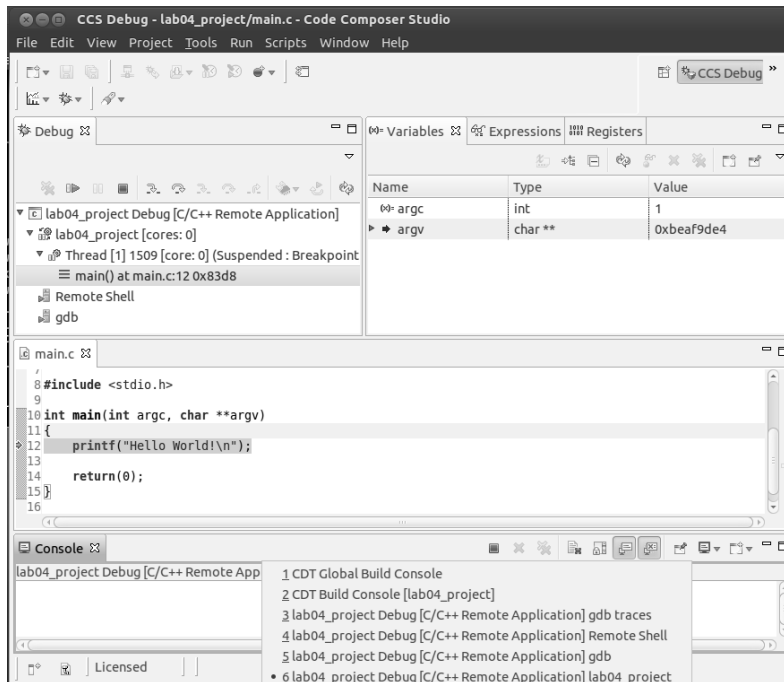
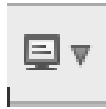
**19. Press the Debug Button to begin Debugging your application.**

Alternately you could press the close button and manually start debugging using  
Run→debug (F11)

**20. After clicking *Debug*, the IDE will switch into the *Debug Perspective*. It will then load the program and execute until it reaches *main()*.**

## 21. View remote terminal

You can view the ssh terminal in which the application is running from the “Display Selected Console” dropdown in the console window



Select the “Remote Shell” terminal and you will see the message “Hello World!” displayed after you step over the printf statement.


## 22. Run through the application once

You may press the start button



or use Run→Resume (F8)

## 23. Restart the application

After each run through of the application, the gdbserver application terminates and a new session must be restarted. This is also what will happen if you press the stop button  or select Run→Terminate (ctrl-F2)

You may restart by pressing the bug button



You can press the button without using the pull-down menu, or by using the pull-down and selecting “ccslab Debug”

Note that if you wish to halt a running application without ending the debug session, you must use the pause button or “Run→Suspend”

## 24. Set some breakpoints, single step, view some variables


You can set a breakpoint by right-clicking on a line of code and selecting “Run→Toggle Breakpoint”, or by pressing (Ctrl-Shift-B). You can also double-click the area just to the left of the code line in the display window.

You can step over a line of code with “Run→Step Over” (F6), or by pressing the step-over icon.

You can run to the next breakpoint with “Run→Resume” (F8) or by pressing the run icon.

You can view a variable by right clicking and selecting “Add Watch Expression”

Of course, this is a very simple hello world program – but feel free to add a variable or two and restart the debugger. Note that any changes made will not take effect until you halt the current debug session, rebuild the application, and then re-launch a new debug session.

In order to make changes, you will need to Press the stop button  or Run→Terminate (ctrl-F2)


Edit the main.c file and then press file→save (ctrl-s)

Rebuild the program with Project→Build all (ctrl-b)

Relaunch the debugger with Run→Debug (F11)

## 25. Exit debugging and return to edit Perspective

Often during development it is more convenient after a quick debugging test to exit the debugger and return to the code editing perspective. Even though it is possible to edit code, rebuild and rerun in the debugging perspective, the editing perspective is generally more useful for making more significant changes. To switch back to the editing perspective,

Press the stop button  or select Run→Terminate (ctrl-F2)

Window→Open Perspective→CCS Edit

# Lab 5: Using GPIO

---

## Introduction

In this lab exercise, you will use file I/O to manipulate the GPIO driver using virtual files. You will begin by testing the GPIO using the Linux terminal, and will then write C programs, first just to turn the GPIO on, and then to blink the GPIO at the rate of 1 Hertz.

Recall that commands that should be executed in the Linux terminal of the host x86 machine (VMware) are shown preceded with the ubuntu prompt:

ubuntu\$

whereas commands that should be executed in the Linux terminal of the AM335x starter kit are shown preceded with the am335x prompt:

am335x\$

## Module Topics

<b>Lab 5: Using GPIO.....</b>	<b>5-1</b>
<i>Module Topics.....</i>	<i>5-2</i>
<i>Boot Board and Test GPIO.....</i>	<i>5-3</i>
<i>Part A, Light LED from C Application .....</i>	<i>5-4</i>
<i>Part B, Blink LED from C Application .....</i>	<i>5-5</i>



## Boot Board and Test GPIO

1. **If you have already booted the board from the SD/MMC interface and have an attached serial console, you can skip this step**

Otherwise power up the AM335x starter kit and login via the serial console. Reference “Lab01: Booting Linux” if needed.

Login as:

User: root

Password: none

2. **View current trigger setting for heartbeat LED**

```
am335x$ cat /sys/class/leds/evmsk:green:heartbeat/trigger
```

3. **Using sysfs, disable heartbeat trigger from heartbeat LED**

```
am335x$ echo "none" > /sys/class/leds/evmsk:green:heartbeat/trigger
```

4. **Disable MMC trigger from MMC LED**

Use the “ls” command to locate the appropriate led within /sys/class/leds

5. **Drive the usr0 GPIO to light the LED**

```
am335x$ echo 1 > /sys/class/leds/evmsk:green:usr0/brightness
```

If you like, you can turn the LED back off by echoing “0” into the same virtual file.

## Part A, Light LED from C Application

**6. If needed, start Code Composer Studio from the Desktop Icon**

**7. Switch to the “lab05\_workspace” workspace**

File→Switch Workspace

The workspace is located at /home/user/oe-layersetup/sources/meta-workshop/recipes-workshop-c/lab05\_workspace

**8. Open “main.c” in the “lab05-student” project**

There are two solution projects as well as the student starter files. You can expand the “lab05-student” project in the Project Explorer view if necessary and then double-click the “main.c” file to open it.

**9. In the “main.c” file of “lab05-student”, use fopen, fprintf, and fclose to accomplish the equivalent of lab05 steps 3-5 from the previous section, using a C program**

Here are the function prototypes for fopen, fprintf and fclose:

```
#include <stdio.h>
FILE *fopen(const char *restrict filename,
const char *restrict mode);
int fprintf(FILE *restrict stream, const char
*restrict format, ...);
int fclose(FILE *stream);
```

Accepted values for the “mode” field of fopen include:

“r”	Read only
“w”	Write only
“rw”	Read and Write

**10. Build your application**

Project→Build All

Or

(Right-click Project)→Build Project

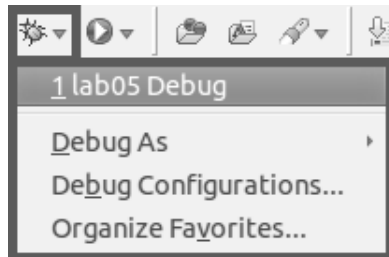
**11. Debug your application**

Debugging has already been configured for this project in the same manner as you set up in Lab 4. You will need to choose “lab05-student” from the available debugging connections.

Use “Run→Debug Configurations...” to bring up the debug configuration

Select the “lab05student” configuration, and then press the “Debug...” button.

Alternately, select “lab05 Debug” from “Run→Debug History” or use the bug pulldown menu



Upon successful completion, you should see user LED 0 light on the AM335x starter kit.

## Part B, Blink LED from C Application

Expand the application by adding a `for` or `while` loop to toggle the user LED 0 between on and off state. A common trick for toggling is to use the exclusive or (XOR) operation:

```
int toggle = 0;

toggle ^= 1;    \\ will change 1 to 0 and 0 to 1
```

After writing the toggle value into the virtual file, you will need to use the `fflush` command; otherwise, the value will be stored in a write buffer but not actually written to the virtual file until the buffer is filled. (`fclose` automatically performs `fflush`, so we didn't have to worry about this in the previous lab.)

You will also want to use the “sleep” command to pause for a second between iterations of the loop; otherwise the GPIO will toggle so quickly you will only see a dimly lit user LED 0. The prototypes for these functions follow:

```
#include <stdio.h>

int fflush(File *stream);

#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

The solution to this exercise (in the “lab05-solution-b” project) also uses signal handling to exit from the while loop when “ctrl-c” is pressed. When debugging with Code Composer Studio, using signals to end a program is less of a concern since CCS may be used to halt the program when desired. Signal handling will be discussed in the next chapter.

(Page intentionally left blank)

# Lab 6: Serial Driver

---

## Introduction

In this lab exercise you will control the AM335x UART via the Linux terminal as well as a C program. The UART in question is an actual RS-232 UART; however, the FTDI USB bridge chip on the AM335x starter kit captures the output of this UART and converts it to a Serial-over-USB interface.

Part B demonstrates writing of formatted strings to the UART, and in part C, you will examine the code used to configure the serial port for 115,200 baud, 8-bit operation with no stop bits and 1 parity bit.

Recall that commands that should be executed in the Linux terminal of the host x86 machine are shown preceded with the ubuntu prompt:

```
ubuntu$
```

whereas commands that should be executed in the Linux terminal of the AM335x starter kit are shown preceded with the am335x prompt:

```
am335x$
```

## Module Topics

<b>Lab 6: Serial Driver .....</b>	<b>6-1</b>
<i>Module Topics.....</i>	<i>6-2</i>
<i>Boot Board and Test SSH.....</i>	<i><b>Error! Bookmark not defined.-Error! Bookmark not defined.</b></i>
<i>Remove Serial Console from AM335x Starter Kit.....</i>	<i>6-3</i>
<i>Part A: Simple UART Driver Hello World .....</i>	<i>6-4</i>
<i>Part B: Optional Challenge – Variables over UART.....</i>	<i>6-6</i>
<i>Part C: Examination Lab – UART Configuration .....</i>	<i>6-7</i>
<i>Restore Serial Console on AM335x Starter Kit .....</i>	<i>6-8</i>

# Remove Serial Console from AM335x Starter Kit

The Linux console that is connected over the UART-over-USB connection on the AM335x is the only serial port that is exposed from the board. If we wish to experiment with using this UART via Linux drivers from an application, we will need to disable the console.

**1. Boot the AM335x starter kit (if necessary)**

**2. Open CCSv6 and switch to the lab06-workspace**

File→Switch Workspace

The workspace is located at:

/home/user/oe-layersetup/sources/meta-workshop/  
recipes-workshop-c/lab06-workspace

**3. Open the remote system view**

Window→Show View→Other...

Expand the “Remote Systems” folder and select the “Remote Systems” view from inside.

**4. Locate “/etc/inittab” within the am335x.gigether.net Remote System View**

Expand the “am335x.gigether.net” connection and then descend the tree through:  
am335x.gigether.net→Sftp Files→Root→/→etc→inittab

Double-click the “inittab” file within the view or right-click and select open.

The file will open within CCS even though it resides on the AM335x Starter Kit!

**5. Edit the inittab file and comment out the serial console setup**

```
ubuntu$ gedit inittab
```

Within the editor, you will need to find the line (should be line number 31):

```
S:2345:once:/sbin/getty 115200 tty00
```

And comment it out by placing a hash (#) at the beginning of the line:

```
# S:2345:once:/sbin/getty 115200 tty00
```

When you are done, save the file and exit using

File→save (ctrl-s)

**6. Reboot the AM335x starter kit**

You may use the push button on the underside of the main board closest to the power supply barrel connector. If your serial port is still live, you will see feedback from u-boot and the booting kernel, but at the end of the boot process, you should not see a login. You will still be able to login to the AM335x using ssh or a like terminal program, however, once it boots.

## Part A: Simple UART Driver Hello World

### 7. Write a program to send “Hello World” over the UART

You should have the following starter file in your lab06-student project:

```
main.c
1#include <stdio.h>
2#include <unistd.h>
3#include <string.h>
4#include <fcntl.h>
5
6int main( void )
7{
8
9
10}
11|
```

### 8. You will need to use the driver calls “open,” “write” and “close” to access the UART driver.

The device node corresponding to the UART that is exposed over the USB connection is “/dev/ttyO0” (this is “tty”, the capitol letter “O”, then the number 0)

Here are a few function prototypes that may help you write your program.

```
#include <unistd.h>

// open returns a file descriptor.
// Accepted flags are O_RDONLY, O_WRONLY, O_RDWR

int open(const char *pathname, int flags);

// count in bytes. Returns number of bytes successfully written

size_t write(int fd, const void *buf, size_t count);

// close returns 0 on success

int close(int fd);
```

Also, you may wish to use the “sizeof” C macro, which will return the size of a variable (or array or struct) in bytes.



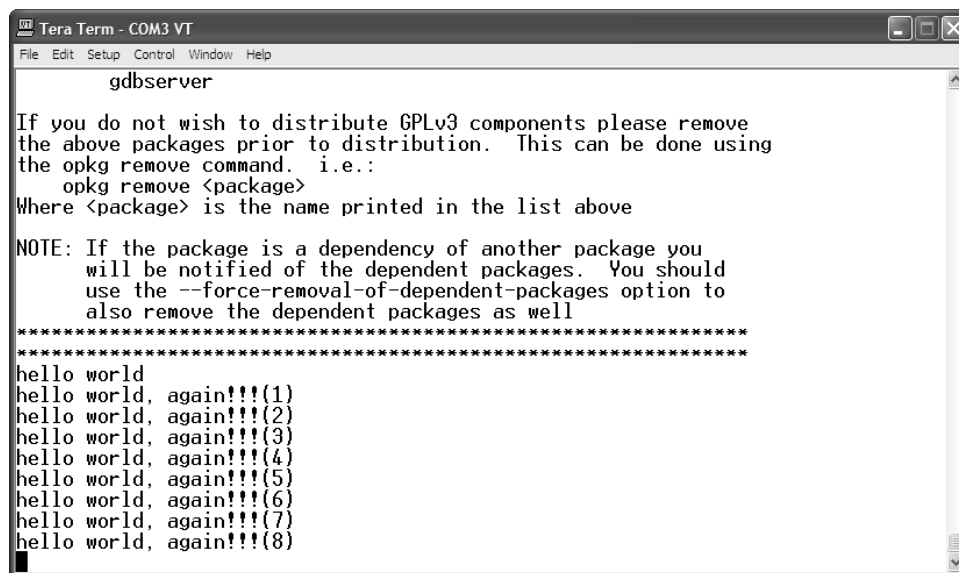
**9. Debug your “Hello World” program**

Be sure to use the “lab06-student” debug configuration, and make sure the serial port terminal (minicom) that you opened in step **Error! Reference source not found.** is still connected to the AM335x starter kit. If the program executes successfully, you should see your message appear on this terminal.

## Part B: Optional Challenge – Variables over UART

Think you got what it takes? This is an optional lab that demonstrates how you can send formatted strings over the UART.

The challenge is to generate something like the following on your UART:



```

Tera Term - COM3 VT
File Edit Setup Control Window Help

gdbserver

If you do not wish to distribute GPLv3 components please remove
the above packages prior to distribution. This can be done using
the opkg remove command. i.e.:
    opkg remove <package>
Where <package> is the name printed in the list above

NOTE: If the package is a dependency of another package you
      will be notified of the dependent packages. You should
      use the --force-removal-of-dependent-packages option to
      also remove the dependent packages as well

*****
hello world
hello world, again!!!(1)
hello world, again!!!(2)
hello world, again!!!(3)
hello world, again!!!(4)
hello world, again!!!(5)
hello world, again!!!(6)
hello world, again!!!(7)
hello world, again!!!(8)

```

### Hint:

The function “snprintf” works pretty much exactly like “printf,” except that it writes to a string instead of writing to the terminal. There is also a function “sprintf,” but the function “snprintf” is preferred because it writes at most “n” bytes to the output string, which guarantees you will not overflow. Here is the function prototype:

```
int snprintf(char *str, size_t size, const char *format, ...);
```

An example to write the value of “j” into the string “iteration”

```

char iteration[10];

snprintf(iteration, 10, "%d", j);

```

## Part C: Examination Lab – UART Configuration

This lab demonstrates configuration of the UART using the “termios” library. You will not see a difference in how the lab executes, because Linux has already configured the UART with the proper settings; however, it is good practice to always initialize the port in such a manner instead of relying on outside configuration files.

### 10. Execute the “lab06-solution-c” debugging configuration

Because the solutions are maintained on a git repository, the source code for main.c is managed by OpenEmbedded and is not immediately available for viewing by CCS. However, by launching the debugger, CCS will locate the source file (within the OpenEmbedded rootfs build for the AM335x) and you will be able to see the solution source.

### 11. Note the “configure\_serial” function at the beginning of main.c

The “termios” library is built on top of the low-level “ioctl” function for configuration of a standard driver. This routine uses bit manipulation (clear and set operations) to configure the “terminal\_setting” variable to 115,200 Baud, 8-bit operation with no stop bits and 1 parity bit.

### 12. Execute the application

You will see the same output as in Part B since the configuration of this lab is simply reconfiguring the serial port to what were already its default settings. If you wish, you can cut and past the “configure\_serial” function from the solution main.c into the “lab06-student” project and see the effect of changing the settings.

## Restore Serial Console on AM335x Starter Kit

Though you could complete the remainder of workshop labs using an ssh terminal, it is handy to have the UART terminal available.

1. **Boot the AM335x starter kit (if necessary)**

2. **Open CCSv6 and switch to the lab06-workspace**

File→Switch Workspace

The workspace is located at:

```
/home/user/oe-layersetup/sources/meta-workshop/  
recipes-workshop-c/lab06-workspace
```

3. **Open the remote system view**

Window→Show View→Other...

Expand the “Remote Systems” folder and select the “Remote Systems” view from inside.

4. **Locate “/etc/inittab” within the am335x.gigether.net Remote System View**

Expand the “am335x.gigether.net” connection and then descend the tree through:

```
am335x.gigether.net→Sftp Files→Root→/→etc→inittab
```

Double-click the “inittab” file within the view or right-click and select open.

The file will open within CCS even though it resides on the AM335x Starter Kit!

5. **Edit the inittab file and comment out the serial console setup**

```
ubuntu$ gedit inittab
```

Within the editor, you will need to find the line (should be line number 31):

```
#S:2345:once:/sbin/getty 115200 ttyO0
```

And uncomment it by removing the hash (#) at the beginning of the line:

```
S:2345:once:/sbin/getty 115200 ttyO0
```

When you are done, save the file and exit using

File→save (ctrl-s)

6. **Reboot the AM335x starter kit**

You may use the push button on the underside of the main board closest to the power supply barrel connector. You will see feedback from u-boot and the booting kernel, and at the end of the boot process, you will now see a login prompt again.

# Lab 7: Linux Scheduler

---

## Introduction

In this lab exercise you will explore the Linux Scheduler using POSIX threads (pthreads) and semaphores. You will create threads that print a message indicating which thread is running. In the first exercise, you will use standard time-slicing threads without semaphores. In the next exercise you will use realtime threads without semaphores, and in the final lab, you will use realtime threads with semaphores.

By examining the output of these applications, you will see firsthand the effect of using time slicing versus realtime threads and the application of semaphores.

# Module Topics

- Lab 7: Linux Scheduler .....7-1**
  - Module Topics..... 7-2*
  - A. Creating a POSIX thread ..... 7-3*
  - B. Real-time Threads ..... 7-7*
  - C. Using Semaphores..... 7-8*

## A. Creating a POSIX thread

1. If needed, start Code Composer Studio from the desktop icon
2. Switch to the “lab07-workspace” workspace  
File→Switch Workspace

The workspace is located at:

```
/home/user/oe-layersetup/sources/meta-workshop/  
recipes-workshop-c/lab07_workspace
```

3. Expand the “lab07-student” project
4. Examine “thread.c”

You can double-click the file in the explorer window or (right-click)→open

This file defines a single function, which is a template for launching a POSIX thread.

```
int launch_pthread( pthread_t *hThread_byref,  
                   int type,  
                   int priority,  
                   void *(*thread_fxn)(void *env),  
                   void *env )
```

The variables used are as follows

`pthread_t *hThread_byref`: this is a pointer to a POSIX thread handle. This is equivalent to passing the handle by reference (as opposed to pass by copy.) The handle pointed to will be overwritten by the `pthread_create` function so that it is effectively used as a return value.

`int type`: `REALTIME` or `TIMESLICE` as `#define`'d in `thread.h`

`int priority`: 1-99 for `REALTIME` thread or 0 for `TIMESLICE`

`void *(*thread_fxn)(void *env)`: this is a pointer to a function, where the function takes a single (void \*) argument and returns a (void \*) value. This is a pointer to the function that will be the entry point for the newly created thread. In C, a pointer to a function is just the name of the function. The entry point for a POSIX thread must be a function with this prototype. A (void \*) pointer is like a skeleton key – any pointer type may be passed through a (void \*) argument. In order for such a pointer to be referenced within the function, however, it must be type cast.

`void *env`: this is the argument that will be passed to the thread function upon entry into the newly created POSIX thread.

5. Open `main.c`

## 6. Examine the “thread\_fxn” template

```
/* Global thread environments */
typedef struct thread_env
{
    int quit;        // Thread will run as long as quit = 0
    int id;
    sem_t *mySemPtr;
    sem_t *partnerSemPtr;
} thread_env;

thread_env thread1_env = {0, 1, NULL, NULL};
thread_env thread2_env = {0, 2, NULL, NULL};

/* Thread Function */
void *thread_fxn( void *envByRef )
{
    thread_env *envPtr = envByRef;

}
```

The thread function takes the standard (void \*) argument; however, note that it type casts this pointer as a (thread\_env \*). The thread environment type is defined just above and contains four elements. By passing a pointer to this structure, you are effectively passing these four elements as parameters to the function.

## 7. Write the thread function

For this first stage of the lab, you will only use the “quit” and “id” fields of the environment structure. Your thread function should have three phases:

1. Print a message to stdout (will be printed to terminal) indicating that the thread has been entered. Be sure to indicate the thread ID (envPtr->id) in this message.
2. Enter a loop that will repeat as long as the quit variable (envPtr->quit) is zero. Inside this while loop, print a message to indicate you are inside the loop, again indicating the thread ID, and then enter a spin loop to pause before the next message (or else your terminal will quickly become flooded with messages!) A good delay value is:

```
for(i=50000000; i > 0; i--);
```

**Note: \*Do not\*** use the “sleep” function. It is important for the lab that this is an actual spin loop, even though that is not good programming!

3. After exiting the while loop, print a final message to indicate that the thread is exiting (include the thread ID in the message) and then return the thread ID as the return value of the function. Note: you do not need to create a return structure. Since both pointers and ints are 32-bit on this architecture, you may cheat and simply recast the ID as a (void \*):

```
return (void *)envPtr->id;
```



**8. Write the main function**

The main function should have the following 5 phases:

1. Print a message indicating you are launching thread 1, then launch this new pthread using the “launch\_pthread” function defined in thread.c. Store the handle to the newly created thread in the “thread1” variable, and pass the “thread1\_env” environment structure. Be sure to launch as a TIMESLICE thread.
2. Print a message indicating you are launching thread 2, then launch this new pthread using the “launch\_pthread” function defined in thread.c. Store the handle to the newly created thread in the “thread2” variable, and pass the “thread2\_env” environment structure. Be sure to launch as a TIMESLICE thread.
3. Print a message to indicate that the application threads have started, then sleep the main thread for 10 seconds using:  
`sleep(10);`
4. Change the quit field of the thread1\_env and thread2\_env environment structures to 1.
5. Use pthread\_join to halt the main thread until both thread1 and thread2 have exited. Be sure to capture the return values of these threads in the thread1Return and thread2Return variables. Print a message indicating which threads have exited using the thread1Return and thread2Return variables (Recall that the return value of the thread\_fxn is the thread ID).

**9. Build the lab07\_pthread\_lab project**

In the project explorer, (right-click lab07\_pthread\_lab)→Build Project  
or  
Project→Build All (ctrl-B)

**10. Debug any build errors**

Don't forget your two resources for lab writing:

(right-click lab07\_pthread\_lab)→Compare With→Branch, Tag or Reference...

Local→solution\_A

Or ask your instructor for help.

**11. Once the program has built, launch the “lab07-student” debug session**

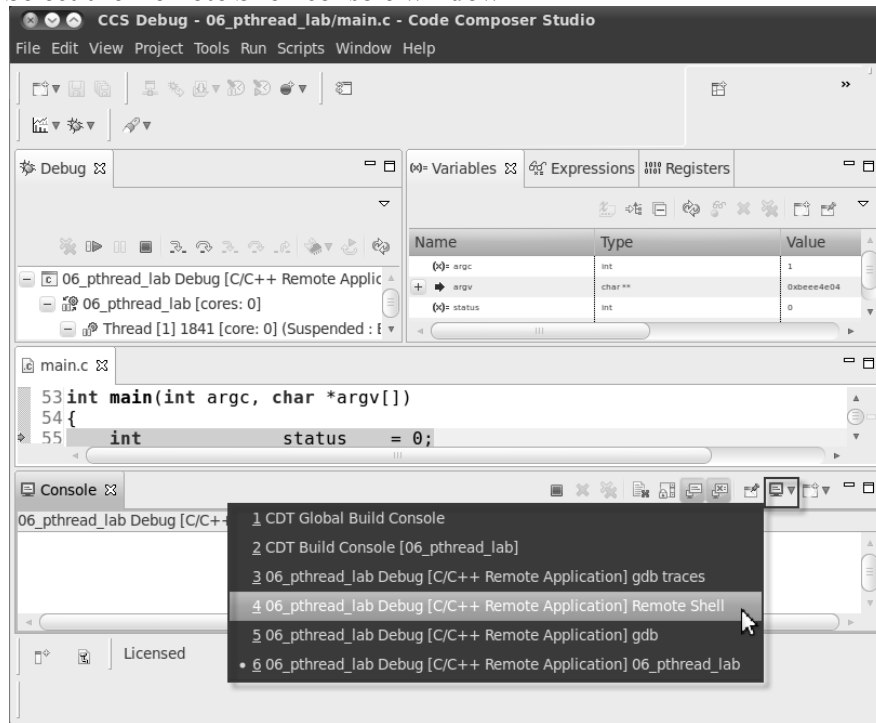
You should be taken to main

```

53 int main(int argc, char *argv[])
54 {
55     int status = 0;
56     pthread_t thread1, thread2;
57     void *thread1Return, *thread2Return;
58
59     /* Create a thread for audio */
60     printf( "Creating thread 1\n" );
61
62     if( launch_pthread( &thread1, TIMESLICE

```

## 12. Select the Remote Shell console window



13. Run the program with the resume button (  ) or Run→Resume (F8)

14. You should see something like the following output (you may have to re-select the Remote Shell view after program terminates)

**Note:** Due to non-determinism of Time-Slice scheduling, results may appear slightly different than as in the window below

```
<terminated> 06_pthread_lab C
Remote debugging from host 192.168.1.1
Creating thread 1
Creating thread 2
Entering thread #1
Inside while loop of thread #1

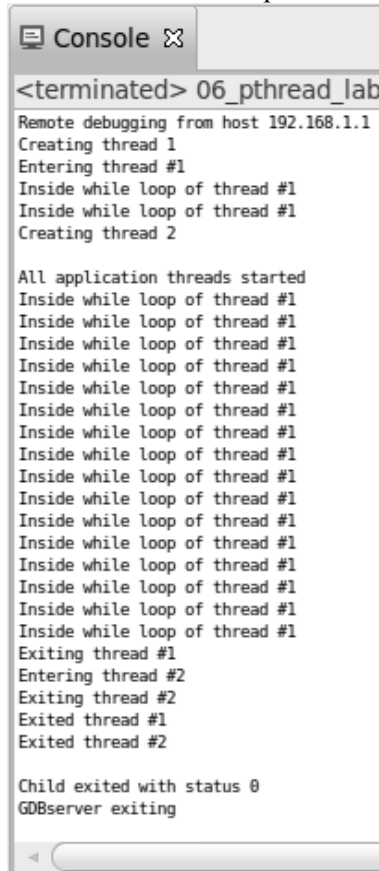
All application threads started
Entering thread #2
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Exiting thread #1
Exiting thread #2
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting
```

## B. Real-time Threads

15. Change thread1 to a REALTIME thread with priority 99
16. Change thread2 to a REALTIME thread with priority 98
17. Rebuild, launch the debugger, and view the Remote Shell output

You should see an output that matches the following:



```
<terminated> 06_pthread_lab
Remote debugging from host 192.168.1.1
Creating thread 1
Entering thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Creating thread 2

All application threads started
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Exiting thread #1
Entering thread #2
Exiting thread #2
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting
```

18. What difference do you see between this output and the TIMESLICE thread output? Why?

---

---

---

---

## C. Using Semaphores

**19. In main.c thread\_fxn, change the while loop for the following functionality:**

1. Do a semaphore wait operation using the thread's "my semaphore" pointer  
(envPtr->mySemPtr)
2. Do a "sleep(1);" after the completion of the semaphore wait operation to slow the system down. (This should replace the for loop that was previously used to delay the system.)
3. Keep the print statement to indicate that execution is inside the while loop of the thread, printing the thread ID
4. Finish the loop by posting the "partner semaphore"  
(envPtr->partnerSemPtr)

**20. In main, create and initialize the semaphores pointed to by**

**"thread1\_env->mySemPtr" and "thread2\_env->mySemPtr"**

You will need to use the "malloc" function to allocate memory for both semaphores, followed by the "sem\_init" function to initialize the semaphores. Be sure to set the initial values for both semaphores to "0."

**21. Initialize "thread1\_env->partnerSemPtr" to point to the same semaphore as "thread2\_env->mySemPtr" and vice-versa**

**22. Create a "trigger" semaphore post in main to post "thread1\_env->mySemPtr" after both threads have been created**

The thread\_fxn has been set up so that both threads will start upon creation with a semaphore wait operation. Since both semaphores were initialized to "0," something needs to kick off one of the threads or nothing will ever happen. Once thread1 is kicked off with the first semaphore post from main, it will post the semaphore for thread2, and from there out there is a one-to-one correspondence between the semaphore wait operations and the semaphore post operations.

An alternative to the triggering post in main would have been to initialize the "thread1\_env->mySemPtr" to an initial value of "1."

**23. Rebuild, launch the debugger, and view the Remote Shell output**

**You should see output that matches the following:**

```
Remote debugging from host 192.168.1.1
Initializing Semaphores
Creating thread 1
Entering thread #1
Creating thread 2
Entering thread #2

All application threads started
Sending trigger sem_post to thread 1
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Exiting thread #2
Inside while loop of thread #1
Exiting thread #1
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting
```

# Lab 8: Berkeley Sockets

---

## Introduction

In this lab exercise you will explore Linux networking on the AM335x starter kit. You will write a Berkeley sockets client application which will send the message “Hello World!” from the AM335x starter kit via an Ethernet connection using Berkeley sockets. Once the client application is working, you will then write a Berkeley sockets server application to run on the host x86 computer to receive the message and print it to the terminal.

Recall that commands that should be executed in the Linux terminal of the host x86 machine are shown preceded with the ubuntu prompt:

```
ubuntu$
```

whereas commands that should be executed in the Linux terminal of the AM335x starter kit are shown preceded with the am335x prompt:

```
am335x$
```

## Module Topics

<b>Lab 8: Berkeley Sockets.....</b>	<b>8-1</b>
<i>Module Topics.....</i>	<i>8-2</i>
<i>A. Berkeley Socket Client.....</i>	<i>8-3</i>
<i>B. Build and Launch the Host Server .....</i>	<i>8-4</i>
<i>C. Server Application.....</i>	<i>8-6</i>

## A. Berkeley Socket Client

1. **Start code composer studio and switch to the “lab08-client-workspace”**

File→Switch Workspace

Be sure that you open the “client” version of this lab and not the “server” version!

2. **Examine main.c from lab08-client-student**

This is an empty program with the necessary header files included. All of the code for the following steps of this section is to be placed within this empty main function.

3. **Open an IP socket using the “socket” function**

The socket command returns a file descriptor, which should be saved in the SocketFD variable.

4. **Request a connection via the SocketFD IP socket using the “connect” function**

The characteristics of the connection are specified via the stSockAddr structure. The connection should be configured to:

family: AF\_INET (IP connection)

address: 192.168.1.1

port: 1100

The address chosen correlates to the (static) IP address of the host computer that the program will connect to. This could have been determined via the “ifconfig” command at a host terminal. The port can be any port that is not already in use; however, the client and server must agree on the port used for the connection. The server program you will use to test the client is configured to use port 1100, a somewhat arbitrary choice.

5. **Write a message into the connected socket using the “write” function**

In the example, we write “Hello World of Ethernet!”

6. **Close the connection with the “shutdown” function**

This is the cleanup for the “connect” function of step 4. At this point, connect could be called a second time using the same socket to establish a new connection.

7. **Close the socket with the “close” function**

This is the cleanup for the “socket” function of step 3.

8. **Build the program and check for any errors, but do not launch the debugger at this time**

The client application will fail if there is no server running on the host computer to accept the connection.

## B. Build and Launch the Host Server

9. **Keep CCS open, but in a separate Linux terminal window on the x86 host, change to the “/home/user/oe-layerssetup/sources/meta-workshop/recipes-workshop-c/lab08-server-workspace/lab08-server-solution-a/files” directory**

Be careful not to change into the “lab08-client-solution-a” directory by mistake.

10. **List the contents of the directory**

There are two files: LICENSE and makefile

11. **Rebuild the application**

```
ubuntu$ make host
```

After a successful completion of the make script, a new executable named “lab08-sever-solution-a-x86” will appear in the directory.

12. **Launch the application with root permissions using the “sudo” command**

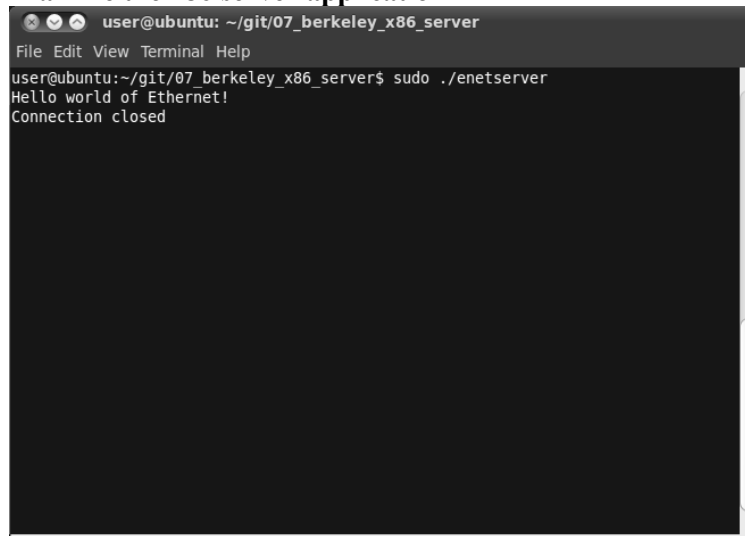
```
ubuntu$ sudo ./lab08-sever-solution-a-x86
```

Until you launch the client application you will not see anything happen. The server program will wait for a connection to be requested by a client, and upon opening a connection will echo any message received to the terminal. If all goes well, you will see the message you wrote into the client appear on this terminal once the client application is run.

13. **Change back to the CCS window and launch the lab08-student application using the CCS debugger**

Be sure to press the “resume” button after launching the debugger as CCS will halt the application at the start of main after it is loaded.

14. **Examine the x86 server application**

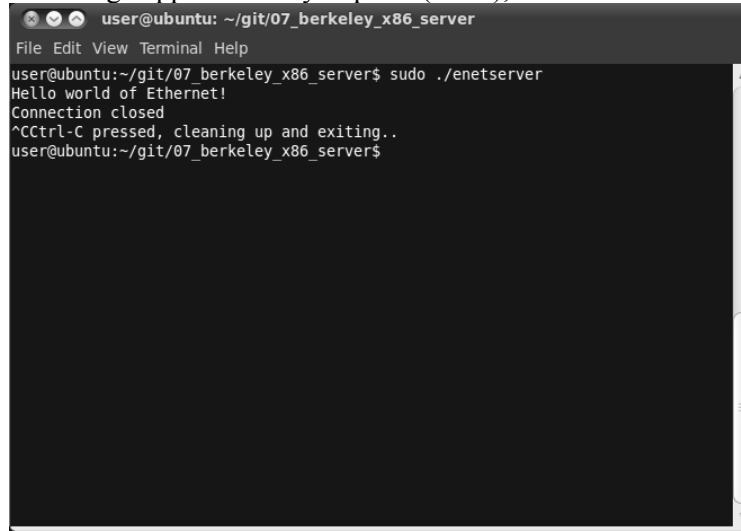


```
user@ubuntu: ~/git/07_berkeley_x86_server
File Edit View Terminal Help
user@ubuntu:~/git/07_berkeley_x86_server$ sudo ./enetserver
Hello world of Ethernet!
Connection closed
```



**15. Exit the x86 server application by pressing (ctrl-c)**

You must have “focus” within the terminal window in order for the (ctrl-c) to take effect. If nothing happens when you press (ctrl-c), use the mouse to select the terminal window.

A screenshot of a terminal window titled "user@ubuntu: ~/git/07\_berkeley\_x86\_server". The window has a menu bar with "File", "Edit", "View", "Terminal", and "Help". The terminal output shows the command "sudo ./enetserver" being executed, followed by "Hello world of Ethernet!", "Connection closed", and "^C Ctrl-C pressed, cleaning up and exiting..". The prompt "user@ubuntu:~/git/07\_berkeley\_x86\_server\$" is visible at the bottom.

```
user@ubuntu: ~/git/07_berkeley_x86_server
File Edit View Terminal Help
user@ubuntu:~/git/07_berkeley_x86_server$ sudo ./enetserver
Hello world of Ethernet!
Connection closed
^C Ctrl-C pressed, cleaning up and exiting..
user@ubuntu:~/git/07_berkeley_x86_server$
```

## C. Server Application

16. If you have not already done so, exit the server application with (ctrl-c)
17. In one of the x86 host computer terminals, change to the “lab08-server-student” files directory

```
ubuntu$ cd /home/user/oe-layersetup/sources/meta-workshop/  
recipes-workshop-c/lab08-server-workspace/lab08-server-student/files
```

18. Run a “make clean” operation to remove the current server binary

19. Open main.c for editing

```
ubuntu$ gedit main.c
```

20. Examine provided signal structure

The main.c starting file has been set up with all of the header files you should require as well as with a registered signal handler for SIGINT (ctrl-c). The signal handler will clean up the current connection and socket by closing ConnectFD (connection file descriptor) and SocketFD (socket file descriptor.) It is important that these file descriptors are properly closed before exiting the application; otherwise they will remain open and will block any further connection via the specified IP address and port.

21. Within the main function, open a socket with the “socket” function and bind it to an IP address and port with the “bind” function

```
family:      AF_INET  
port:        1100  
address:     INADDR_ANY (you could also use 192.168.1.1)
```

22. Ready the socket to accept connections with the “listen” command

23. Create a while loop to accept connections and read data as it comes across

The while loop should have the following stages:

1. Begin the while loop by accepting a connection from the socket. (This function will block until a connection request is made from a client)
2. Create an inner while loop that reads a byte from the connection and writes it to standard out (you can use STDOUT\_FILENO as the file descriptor for standard out.) You should exit from the inner loop when the read command returns “0” for the number of bytes read, which indicates a termination of the connection.
3. After exiting the inner while loop, indicating the connection was closed by the client, close the connection on the server side with the close command. The solution also prints a message to standard out to indicate that the connection was closed at this point.

24. Save the file

25. Rebuild the server executable

```
ubuntu$ make host
```

26. Test your server executable with the known-good client that you wrote in section C

```
ubuntu$ sudo ./lab08-server-student-x86
```

# Lab 09: Framebuffer Driver

---

## Introduction

In this lab exercise you will explore the Linux framebuffer device (FBDEV) driver. This is the low-level driver that controls graphical displays such as LCD screens. The framebuffer driver is so named because it exposes the display bitmap (i.e. the “frame buffer”) as a virtual file. You will begin by using the standard “write” command to write an image into this virtual file. You will then use the more efficient “mmap” command to map this virtual file as an array into the application memory space and manipulate it directly. In the final step of the lab, you will implement a double-buffered system using mmap.

## Module Topics

<b>Lab 09: Framebuffer Driver .....</b>	<b>9-1</b>
<i>Module Topics.....</i>	<i>9-2</i>
<i>Copy-based Framebuffer Application.....</i>	<i>9-3</i>
<i>mmap-based Framebuffer Application .....</i>	<i>9-4</i>
<i>Double-buffered Framebuffer Application.....</i>	<i>9-5</i>

## Copy-based Framebuffer Application

1. **Start Code Composer Studio and switch to the “lab09-workspace” workspace.**
2. **Examine the main.c starter file in the “lab09-student” project.**

A small section of code has been provided to write the value “0” into the sysfs virtual file “/sys/class/graphics/fb0/blank.” This enables the backlight for the display (i.e. disables blanking of the display.)

A number of variable declarations have been provided as well.
3. **Open the “/dev/fb0” device node.**

This is the device node corresponding to framebuffer zero, the default framebuffer device for the system.
4. **Query the framebuffer driver for its fixed screen information.**

You may use the “ioctl” driver function with the “FBIOGET\_FSCREENINFO” command enumeration. For convenience, the ioctl function prototype is provided below:

```
int ioctl(int fd, int request, void *argument);
```
5. **Query the framebuffer driver for its variable screen information.**

You may use the “ioctl” driver function with the “FBIOGET\_VSCREENINFO” command enumeration.
6. **Use nested for loops to write pixel values into the framebuffer bitmap.**

You can determine the number of pixels in a row by using the “fbfix.line\_length” determined in step 4. Note that the line length is always specified as the number of bytes in one row of the bitmap. Each pixel has a four-byte color value, so the number of pixels in a row is the line length divided by four.

You can set the number of rows equal to “fbvar.yres” as determined in step 5.

Each pixel has a 32-bit ARGB color value: 8-bit attribute, 8-bit red, 8-bit green and 8-bit blue. You may color with any scheme you like. The solution creates a 2-d color gradient using:

```
pixel_val = ((j/2) << 8) + (unsigned int) (0.9 * i);
```
7. **Once the nested for loops have set each pixel value, close the framebuffer file descriptor.**

Closing the file descriptor does not erase the contents of the framebuffer. The pixel values you have written will persist within the framebuffer after closing the file descriptor unless they are overwritten.
8. **Build and test your application.**

## mmap-based Framebuffer Application

### 9. Insert the “mmap” function into main.c

This will need to come after the call that opens “/dev/fb0” because you will need the returned file descriptor, and it should also come after the “ioctl” calls for determining fixed and variable screen info as you will need this information to determine the size of the buffer to memory map.

For convenience, the mmap function prototype is provided below:

```
void *mmap(void *addr,    // use "0" (zero)
           size_t len,    // buffer length in bytes
           int prot,      // use "PROT_READ | PROT_WRITE"
           int flags,     // use "MAP_SHARED"
           int fd,        // device file descriptor
           off_t offset); // use "0" (zero)
```

The return value is a user-space (i.e. virtual) pointer to the newly mapped memory buffer.

Most of the parameters that have been provided for you are unlikely to change. Passing a value other than zero in the initial address field allows you to request a specific physical address for the framebuffer to be mapped to, which is rarely used. The “prot” field indicates that the mapped memory can be written and read. The “MAP\_SHARED” flag indicates that the changes will be written to the underlying file (i.e. framebuffer) as opposed to “MAP\_PRIVATE,” which would allow reading of the framebuffer, but would write all changes to a private copy of RAM and never update the framebuffer. The offset field specifies the offset within the framebuffer where the mapping into program RAM will begin.

### 10. Modify the pixel manipulation within your nested for loops to write directly into the memory mapped region.

You will no longer require the “write” driver function. Instead, you can use the memory mapped array to write directly into the framebuffer.

### 11. Build, test and commit.

## Double-buffered Framebuffer Application

The FBDEV driver uses the concept of a virtual display space. Using a virtual display space that is twice as large as the physical display's bitmap, the application will create a "ping" and a "pong" display buffer within the virtual space and use the "FBIO\_PANDISPLAY" ioctl call to toggle between them.

**12. Change the mmap function call in main.c to map a buffer size that is twice as large as the display size.**

Previously the application used line length and height of the display to calculate the size of the display bitmap. By mapping a buffer that is twice this size from the virtual memory space of the framebuffer driver, you will have enough memory to implement a "ping" and a "pong" buffer for the display.

**13. Add an infinite "for" or "while" loop around the nested loops created in step 6.**

You can use Code Composer Studio to halt the application, so there is no issue with creating an infinite loop.

**14. Declare an int named "pingOrPong" and toggle it between "1" and "0" on each iteration of the infinite loop that was set up in step 13.**

You can use:

```
pingOrPong ^= 1;
```

to toggle the value between 1 and 0;

**15. Modify the nested loops from step 6 to offset the array by (pingPong \* buflen/4)**

The formula "(pingPong \* buflen/4)" assumes that you are using a pointer to int and that buflen is specified in bytes. This offset will place the modified pixels in the correct "ping" or "pong" area of the virtual memory space.

**16. After the nested loops complete, update the displayed buffer using the "FBIO\_PAN\_DISPLAY" ioctl function call.**

You will use the "fbvar" structure that you initialized with the "FBIOGET\_VSCREENINFO" ioctl in step 5 as the input to the FBIO\_PAN\_DISPLAY ioctl function call. One of the variable fields in this structure is "fbvar.yoffset," which defines the vertical offset of the display space within the virtual display buffer of the driver.

Use:

```
fbvar.yoffset = pingPong * fbvar.yres;
```

To adjust the offset as appropriate for the ping or pong buffer, and then use:

```
ioctl(fd, FBIO_PAN_DISPLAY, &fbvar);
```

to pan the offset to that location.

**17. Use the "FBIO\_WAITFORVSYNC" ioctl to synchronize to the display.**

"FBIO\_PAN\_DISPLAY" is a hardware-latched function, so it does not actually take effect until the vertical sync of the display. However, for a simple application, it would be possible for the application to update the display twice or even more in between vertical sync events. Proper synchronization requires calling the "FBIO\_WAITFORVSYNC" ioctl after calling "FBIO\_PAN\_DISPLAY" to guarantee that the application does not begin processing the next buffer until the latched "FBIO\_PAN\_DISPLAY" ioctl completes.

Directly after the "FBIO\_PAN\_DISPLAY" ioctl function call, use:

```
ioctl(fd, FBIO_WAITFORVSYNC, NULL);
```

to block the application until the hardware vertical sync event occurs.

**18. Implement the “FBIO\_WAITFORVSYNC” workaround.**

The “FBIO\_WAITFORVSYNC” ioctl has not been integrated into the mainstream Linux fbdev driver, so you will have to use the following definition at the top of main.c (after the header file include statements):

```
#define FBIO_WAITFORVSYNC _IOW('F', 0x20, u_int32_t)
```

Note: be sure not to put a semicolon at the end of this line.

**19. Build and run the application.**

As described above, the application draws exactly the same pixels into the ping and pong buffers, so it may not be obvious that anything has changed. This is a good point to test the code, however, and determine if everything is working properly up to this point.

**20. (Optional) Implement some form of motion.**

If you have extra time, you may want to implement some form of motion in the display, such as horizontal scrolling between iterations of the loop. Motion will demonstrate the necessity of a double-buffered system as a properly implemented double-buffered system will not have “tearing” artifacts as would appear in a single-buffered system attempting motion.



# Lab 10: Using qt

---

## Introduction

The Qtopia (“QT” for short) graphics package is an open-source GUI development platform used in many Linux and Windows applications. This lab exercise will lead you through the creation of a simple “Hello World” project built using an OpenEmbedded Bitbake recipe. You will first test this project by running it within the x86 Linux host environment. One advantage to developing with Qtopia is that the language is device (and even operating system) independent, so that applications written for the AM335x can easily be tested in an x86 environment. You will then rebuild the application for the AM335x starter kit and test the application using Code Composer Studio’s remote debugging capability.

## Module Topics

<b>Lab 10: Using qt .....</b>	<b>10-1</b>
<i>Module Topics.....</i>	<i>10-2</i>
<i>qt Hello World .....</i>	<i>10-3</i>

# qt Hello World

By default, QT Creator is configured to create QT applications for the native environment on to which it is installed. This will be a useful feature, as you will be able to test and debug applications within the Ubuntu x86 environment before deploying to the AM335x starter kit.

**1. Use gedit to open the lab 10 Bitbake files**

```
ubuntu$ cd /home/user/oe-layerssetup/sources/meta-workshop/  
recipes-workshop-qt/lab10-workspace/lab10-student  
ubuntu$ gedit lab10-student_1.0.bb lab10-student_1.0.inc
```

**2. Examine lab10-student\_1.0.bb**

The first line of “lab10-student\_1.0.bb” uses the “require” command to include the “lab10-student\_1.0.inc” file. Note that PN and PV are bitbake-defined variables for the package and

```
_${PN}=lab10-student  
_${PV}=1.0
```

Note the line “inherit qt4e” This line inherits the “qt4e” bitbake recipe class. This class contains task definitions to build a qt project. (The acronym stands for “qt for embedded.”)

The remaining lines are a bit more advanced and not important at this stage. Basically they define the “do\_my\_package\_write\_ipk” function which will force the result of the package to be copied directly into the target root filesystem that is maintained by OpenEmbedded.

Generally one package should not touch the workspace of another package in this way, but this is a hack that allows the package to update the target root filesystem without rebuilding the entire rootfs package each time, which would take a significant amount of time. Recall that OE is not meant to be a development/debug system, it is meant for generating a full OS image, and as such build time optimization is not a priority.

**3. Examine lab10-student\_1.0.inc**

The recipe defines the “SRC\_URI” variable to bring in any source files in the package files directory which have the extensions: “pro,” “c,” “cpp,” or “h,” in addition to bringing in the LICENSE file.

The package then overrides the “do\_install” function in order to pass the “INSTALL\_PATH” variable during the call of “make install.” The use of this variable will be demonstrated in step 5.

The remaining tasks used to build the package are used unchanged from the “qt4e” class definitions.

**4. Start CCS and change to the “lab10-workspace” workspace**

You can use File→Switch Workspace...

Note that the qt projects are maintained in a separate directory from the standard C projects. This is because the Bitbake recipe files are different for the qt projects, and keeping them in separate directories makes it easier to apply a script to generate or update the recipe files.

The full path to the “lab-10” workspace is:

```
/home/user/oe-layerssetup/sources/meta-workshop/recipes-workshop-qt/lab10-workspace
```

**5. Examine lab10-student.pro**

As with any file in the project, you may open by double-clicking within the project explorer window.

This is a standard qt project file. You can see that the project file defines the sources that comprise the project, in this case just the “main.cpp” file. Also, it defines the path to install the application on the target, which is reference to the variable “\$(INSTALL\_PATH)”

This variable is passed to the project during the “do\_install” Bitbake command as was shown in step 3.

**6. Examine the provided main.cpp**

This file is an empty shell containing only the standard “main” entry point for C/C++ applications and the standard qt wrapper, which is the definition of a QApplication variable and the calling of the “exec” task from this object.

As written, the application does not have any qt objects and will simply launch the QT environment with nothing in it.

**7. Fill in the remaining application**

You will need to initialize a QLabel object and then use the “show” task to make the object visible within the qt environment.

**8. Rebuild the application within CCS****9. Once the application builds correctly, test it within the x86 system**

Open a Linux terminal outside of CCS (you can use ctrl-atl-t if you have scope in the Ubuntu desktop.)

```
ubuntu$ cd /home/user/oe-layerssetup/sources/meta-workshop/
```

```
recipes-workshop-qt/lab10-workspace/lab10-student/files
```

```
ubuntu$ make host
```

```
ubuntu$ ls
```

Upon a successful build, you should see the file “lab10-student-x86” in the directory.

**10. Launch qt’s virtual framebuffer**

qt has a virtual framebuffer application that allows you to test embedded graphics projects within the x86 environment. It is part of the “qt4-dev-tools” aptitude package, which has already been installed on your system.

**11. Execute the application within the x86 environment using qt’s virtual framebuffer**

```
ubuntu$ sudo ./lab10-student-x86 -qws -display QVFB:0
```

**12. Return to CCS**

If you closed the application, you may simply re-launch it. Otherwise return to the CCS window.

**13. Examine the debug configuration for “lab10-student”**

Run→Debug Configurations...

**14. View the arguments in the “arguments” tab**

Right-click the “(x)=Arguments” tab in the debug configuration. Note that CCS has been configured to pass the “-qws” argument to the application when it is launched. This is required for embedded qt applications as it launched the “qt window server” on the target.

**15. Launch/Debug the application**

Press the “Debug” button

## Introduction

In this lab exercise you will explore the git tool, an open-source tool for version control. You will create an empty git repository and manipulate test files from the Linux terminal, using the git tool to save “commits” in between changes and then restore previous commits using tags. You will also create a second branch and use the git merge capability.

In the next lab exercise, you will build on this knowledge using the git eclipse plugin to save commits of lab exercises and, if necessary, use the diff utility to compare your project to the solution files, which are committed on a separate “solutions” branch.

## Module Topics

<b>Lab 11: git .....</b>	<b>11-1</b>
<i>Module Topics.....</i>	<i>11-2</i>
<i>Create a New git Repository .....</i>	<i>11-3</i>
<i>Explore a Workshop git Repository .....</i>	<i>11-8</i>

# Create a New git Repository

The lab exercise will begin by creating a new repository.

Note that the commit and tag commands use the “-m” option to provide commit and tag messages. If the “-m” option is excluded or specified incorrectly, git will automatically route you into a text editor to write the commit or tag message. If you are routed into the text editor and are unable to save your message and exit, ask your instructor for help.

**1. Make a directory named “test\_repo” at /home/user/**

```
ubuntu$ cd /home/user/  
ubuntu$ mkdir test_repo
```

**2. Initialize the test\_repo directory with “git init”**

```
ubuntu$ cd test_repo  
ubuntu$ git init
```

**3. Create a text file to save**

There are many ways to do this. Let’s try out a Linux redirect (>)

```
ubuntu$ echo "File initialization" > testfile
```

Note that there is a single greater-than sign, which will overwrite this value into the file (as opposed to appending.)

**4. Verify contents of testfile**

```
ubuntu$ cat testfile
```

**5. Commit testfile**

```
ubuntu$ git add testfile  
ubuntu$ git commit -m "commit: Initial file"
```

**6. Create a tag for your initial commit**

```
ubuntu$ git tag v1.0 -m "tag: Initial file"
```

**7. Modify testfile**

```
ubuntu$ echo "Mod 1, master branch" >> testfile
```

Note that the usage of two greater-than signs indicates the Linux redirect will append the echoed message into testfile.

**8. Verify contents of testfile**

```
ubuntu$ cat testfile
```

**9. Commit testfile**

We can use the “-a” flag as a shortcut since “testfile” has already been added to the repository index.

```
ubuntu$ git commit -a -m "commit: Mod 1, master branch"
```

**10. Checkout v1.0 of the repository**

```
ubuntu$ git checkout v1.0
```

git will display a message indicating that you are in “Detached HEAD” state. This means that your current position within the repository is not at the end of a branch. This is important because git does not allow you to modify a commit within a branch since this would affect every commit that comes after it in the branch.

As the message indicates, if you want to save any changes you make to the files in this commit, you will need to create a new branch from the commit point. This will then place the current position (i.e. HEAD pointer) at the end of that (newly created) branch, allowing you to create commits on the new branch.

Since you will not be modifying the files at this commit point, there is no need for you to create a new branch.

**11. Verify contents of testfile**

```
ubuntu$ cat testfile
```

How does this compare to the state of the test file when viewed in step 8?

**12. Check out the v1.1 tag**

```
ubuntu$ git checkout v1.1
```

git displays the same “detached HEAD” state message as before. Why is this, when the v1.1 commit is the last commit on the branch?

The answer is a subtle but important distinction. There is a difference between checking out a branch and checking out the last commit on the branch. This is because all commits, even the last commit on a branch are unchangeable. git will only allow you to make modifications to a branch when you checkout the branch.

**13. Check out the master branch**

```
ubuntu$ git checkout master
```

Now there is no “detached HEAD” message. Again, even though the files are identical to the state they were in when the “v1.1” tag was checked out, the state of the repository is different.

**14. Modify testfile**

```
ubuntu$ echo "Mod 2, master branch" >> testfile
```

**15. Commit testfile**

**16. Create a tag “v1.2” for your latest commit with the message “tag: Mod 2, master branch”**

**17. Print the list of this repository’s commits and tags**

```
ubuntu$ git tag
ubuntu$ git log
```

**18. Determine which commit node each tag corresponds to**

Begin by listing the current tags

```
ubuntu$ git tag
```

You can then print out the hash corresponding to the commit node that a given tag references

```
ubuntu$ git rev-parse --verify v1.0^{commit}
```

Finally, you can locate the commit within the log by its hash

```
ubuntu$ git log
```

**19. Print the current testfile**

```
ubuntu$ cat testfile
```

**20. Check out the previous commit**

```
ubuntu$ git checkout HEAD^
```

Note: You will get a “detached HEAD” message as explained in step 10.



The caret (^) symbol in a git context is used to refer to the commit previous to a given reference. Recalling that “HEAD” references the user’s current position within the git tree, “HEAD^” would be the last commit. The caret symbol may be stacked, i.e. “HEAD^^” and it may also be used in conjunction with other references such as tags. (“v1.1^”) Note that the caret will always reference the previous commit, so that “v1.1^” will only take you to the “v1.0” tag if there are no untagged commits inbetween.

**21. Print the testfile**

**22. Create a new branch off of the v1.0 tag**

```
ubuntu$ git branch test
```

**23. Print a list of branches**

```
ubuntu$ git branch
```

You should see:

```
* (no branch)
```

```
master
```

```
test
```

The asterisk next to “no branch” indicates that you are currently in “detached HEAD” state.

**24. Move to the test branch**

```
ubuntu$ git checkout test
```

**25. Print a list of branches**

```
ubuntu$ git branch
```

You should now see:

```
master
```

```
* test
```

**26. Create a new file named testfile2 with the following text:**

```
ubuntu$ echo "Mod 1, test branch" > testfile2
```

**27. Commit your change (without tag)**

Don’t forget to “git add” testfile2 into the index!

**28. Modify testfile and use “git add” to add it for staging**

**29. Modify testfile2 again, but do not use “git add” to stage it**

**30. Create a new file named “testfile3,” and do not use “git add” to add it to the index.**

### 31. View the current status of the git repository

```
ubuntu$ git status
```

you should see:

```
# On branch test
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   testfile
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be
#    committed)
#   (use "git checkout -- <file>..." to discard changes in
#    working directory)
#
#       modified:   testfile2
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
#    committed)
#
#       testfile3
```

### 32. Commit your change and tag as “v2.0”

### 33. Print the log

Note that the log shows commits only for the current branch

### 34. (Attempt to) Checkout the master branch

You should get an error indicating that your local changes to “testfile2” would be overwritten by checking out the master branch. This is because in step 32, only the changes to “testfile” were added to commit “v2.0”

There are two solutions when this occurs. The first solution would be to create a new commit and “git add testfile2” In some cases, however, you might want to (permanently) discard whatever remaining local changes have been made and perform the checkout.

### 35. Discard changes to current branch

```
ubuntu$ git reset --hard
```

### 36. Checkout the master branch

```
ubuntu$ git checkout master
```

### 37. Do a list command (“ls”)

You should see “testfile” and “testfile3” but no “testfile2.”

“testfile” is present because it is present in this commit node of the git repository.

“testfile3” is present because, even though it is not present in this commit node of the git repository, it is an untracked file, indicating that it is not affected by the “git checkout” command.

“testfile2” is not present because it is not present in this commit node, but it is a file that is tracked by the git repository (i.e. it is in the git index).

**38. Merge the “test” branch into the “master” branch**

```
ubuntu$ git merge test
```

Note: if you have made even a slight deviation from the instructions, you may get a merge conflict message indicating that the changes in the two branches are such that git does not know how to merge them. This is very common when merging branches (more common in fact than merging without a conflict!)

git requires you to resolve merge conflicts by hand. The tool will mark the merge conflicts in the files where they occur and the user then edits the files with a text editor to resolve them. This is an advanced topic and will not be covered in the workshop.

**39. Do another list command**

The master branch now has “testfile,” “testfile2,” and “testfile3.” Try printing the contents of testfile2 with the “cat” command.

**40. View your repository graphically using gitk**

```
ubuntu$ gitk --all
```

## Explore a Workshop git Repository

The git tool is used for version management of software, with principles similar to other revision control systems such as CVS, subversion or clearcase.

This workshop introduces git because it is the most commonly used version management tool in the Linux community, having been developed by Linus Torvalds (the original developer of the Linux kernel) specifically for version management of the Linux kernel.

Not only is the Linux kernel managed on a git repository, but the majority of source code in the GNU community, i.e. the source code that forms full distributions of the Linux operating system, is managed on git repositories as well.

Additionally both the OpenEmbedded recipe files that specify Texas Instruments' Arago Linux distribution as well as the Bitbake overlay used to modify the distribution for this workshop ("meta-workshop") are managed on git repositories. This lab exercise will begin with an exploration of the "meta-workshop" git repository. At the end of the lab, you will create a new branch to store any changes you make to the workshop Bitbake recipes during the course of the workshop.

### 1. Locate the "meta-workshop" git repository

```
ubuntu$ cd /home/user/oe-layerssetup/sources/meta-workshop
```

### 2. Display the git repository branches

```
ubuntu$ git branch
```

The output of this command shows that there is only one branch for this repository, named "master." The asterisk indicates that this is the currently checked-out branch.

### 3. Search the git repository tags

```
ubuntu$ git tag
```

The output of this command will show you a list of commits that have been tagged. This repository tags certain commits with a version number, which is a common use of tags, although not the only use. In this repository, the version numbers are followed by an indication of which sdk version the layer is meant to overlay.

### 4. View the commit log between v1.0\_sdk07.01.00.00 and v1.2\_sdk07.01.00.00

Not all commits are tagged. If you wish to see the details of every commit that was made between two tags use

```
ubuntu$ git log v1.0_sdk07.01.00.00..v1.2_sdk07.01.00.00
(note: two periods between tags)
```

You may scroll through the log with the up and down arrow keys. Press "q" to quit.

### 5. Create a new branch to use for workshop development

Note: the examples will assume a branch named "student;" however, you may name this branch whatever you wish.

```
ubuntu$ git branch student
```

**6. Check out your development branch**

```
ubuntu$ git checkout student
```

From this point onward, any changes that you make to the workshop recipe files will be stored in the “student” branch. This will allow you to revert back to the workshop start files if necessary.

(Page intentionally left blank)

# Workshop Setup Guide

---

## Introduction

This document contains information for setting up an Ubuntu 12.04 host computer to run the lab exercises of the “Introduction to Embedded Linux One-Day Workshop.”

It consists of 4 required sections:

- Installing Ubuntu 12.04
- Installing Code Composer Studio
- Installing Lab Files
- Configuring Ubuntu Static IP

After completing these installation steps, you will have everything needed to run the lab exercises on your system.

Additionally, a number of steps were taken to make the environment more user friendly (“Installing Gnome3 and Standard Scrollbars”) and to set up the lab files toolchain and target filesystem. These comprise the three optional sections. There is no need to go through the optional sections in order to run the lab exercises, but if you would like to know the steps that were required to set up that portion of the lab environment, the steps are shown in these optional sections.

The lab files that you will need to install are located on the workshop wiki page at:

[http://processors.wiki.ti.com/index.php/Introduction\\_to\\_Linux\\_One-Day\\_Workshop](http://processors.wiki.ti.com/index.php/Introduction_to_Linux_One-Day_Workshop)

## Chapter Topics

<b>Workshop Setup Guide .....</b>	<b>1-1</b>
<i>Installing Ubuntu 12.04 .....</i>	<i>1-3</i>
<i>Installing Code Composer Studio v.5.3.0.00090 .....</i>	<i>1-4</i>
<i>Installing Lab Files .....</i>	<i>1-6</i>
<i>Configuring Ubuntu Static IP.....</i>	<i>1-8</i>
<i>(Optional) Installing Gnome3 and Standard Scrollbars .....</i>	<i>1-9</i>
<i>(Optional) Installing Angstrom Cross-compile Tools.....</i>	<b>Error! Bookmark not defined.-Error! Bookmark not defined.</b>
<i>(Optional) Modifying Angstrom Filesystem.....</i>	<b>Error! Bookmark not defined.-Error! Bookmark not defined.</b>



# Installing Ubuntu 12.04

**Note:** Currently the Eclipse IDE platform upon which Code Composer Studio is based uses 32-bit libraries. It is recommended that you install the 32-bit desktop Ubuntu 12.04 even if you have a 64-bit machine.

There are many tutorials available for installing Ubuntu, so this section will not go through great detail on the actual installation; however, it provides information for removing the user password and setting automatic login, as is done in the workshop image.

## 1. Begin by downloading an Ubuntu 12.04 image and burning onto an installation disk.

Please see above. It is recommended that you install the 32-bit version of Ubuntu even if you are using a 64-bit machine. There are tools that allow installation via an installation CD (requires a CD burner to create the installation disk) as well as installation via a USB mass-storage device. Either method is fine.

## 2. Install Ubuntu 12.04 on your computer.

Other versions of Ubuntu may also work, but version 12.04 is what has been tested for this workshop.

If you select “automatic login” on the user setup screen, you can skip step 4.

Be sure to write down the password that you set! The following steps will show you how to remove the password, but you will need to know the old one.

## 3. Open a terminal

ctrl-alt-t

## 4. Select automatic login (If you forgot to select in step 2)

```
# sudo gedit /etc/lightdm/lightdm.conf
```

Enter the password from step 2 when prompted.

Add the following four lines under the section header “[SeatDefaults]”

```
autologin-guest=false
autologin-user=user
autologin-user-timeout=0
autologin-session=lightdm-autologin
```

## 5. Allow null passwords for sudo

```
# sudo gedit /etc/sudoers
```

Enter the password from step 2 if prompted.

Locate the line that reads:

```
%admin ALL=(ALL) ALL
```

And change to read

```
%admin ALL=(ALL) NOPASSWD: ALL
```

## 6. Allow null passwords for authorization (i.e. login)

```
# sudo gedit /etc/pam.d/common-auth
```

Locate the line that contains “nullok\_secure” and change “nullok\_secure” into just “nullok”

## 7. Remove user password

```
# sudo passwd -d user
```

## 8. Reboot to test

```
# sudo shutdown -h now
```

When Ubuntu reboots, open a terminal and try the sudo command:

```
# sudo ls
```

If everything has worked correctly, the list operation should complete without prompting for a password.

# Installing Code Composer Studio v.6.0.1.00040

You can download CCS6 from:

<http://processors.wiki.ti.com/index.php/CCSv6>

and selecting the “Linux” download button.

Also, because older versions of CCS are not always archived, the exact version used in the workshop is also available from the wiki page.

**9. Install required libraries**

```
ubuntu# sudo apt-get update
ubuntu# sudo apt-get install libgnomevfs2-0
ubuntu# sudo apt-get install liborbit2-dev
```

**10. Download CCS from the above listed link**

**11. Add executable permission to the installer**

```
# chmod a+x ccs_setup_linux32.bin
```

**12. Run the installer program**

```
# ./ccs_setup_linux32.bin
```

**13. Read and accept the license agreement.**

**14. Accept the default install directory: /home/user/ti**

**15. Select “Sitara 32-bit ARM Processors” from the Processor Support window, and in the submenu select “GCC ARM Compiler”**

You should not select “TI ARM Compiler.” This compiler does not have Linux support and is used when Sitara processors are run without the Linux operating system. The “GCC Arm Compiler” will not actually be used to generate code because OpenEmbedded will be used instead, but this compiler is used to parse source files.

**16. In “Select Emulators” window, leave the defaults of XDS100 and XDS200 support.**

These emulators are supported under the free license.

**17. Leave the default for “App Center” (no packages installed)**

Though PRU and Linux Development Tools are interesting packages for AM335x development, they will not be used in this workshop.

**18. Press “Finish” to begin the installation**

**19. Select “Create Desktop Shortcut” on final screen (Should be selected by default.)**

**20. Change desktop shortcut to a custom script**

Right-click the shortcut and select properties  
Change “command:”

```
From: /home/user/ti/ccsv6/eclipse/ccstudio
To: /home/user/ti/ccsv6/eclipse/my_ccstudio.sh
```

**21. Create “myccstudio.sh” launch script**

```
# gedit /home/user/ti/ccsv6/eclipse/my_ccstudio.sh
```

Enter the following:

```
export PATH=/home/user/gcc-linaro-arm-linux-gnueabi-4.7-2013-03-
20130313_linux/bin:$PATH
/home/user/ti/ccsv6/eclipse/ccstudio
```

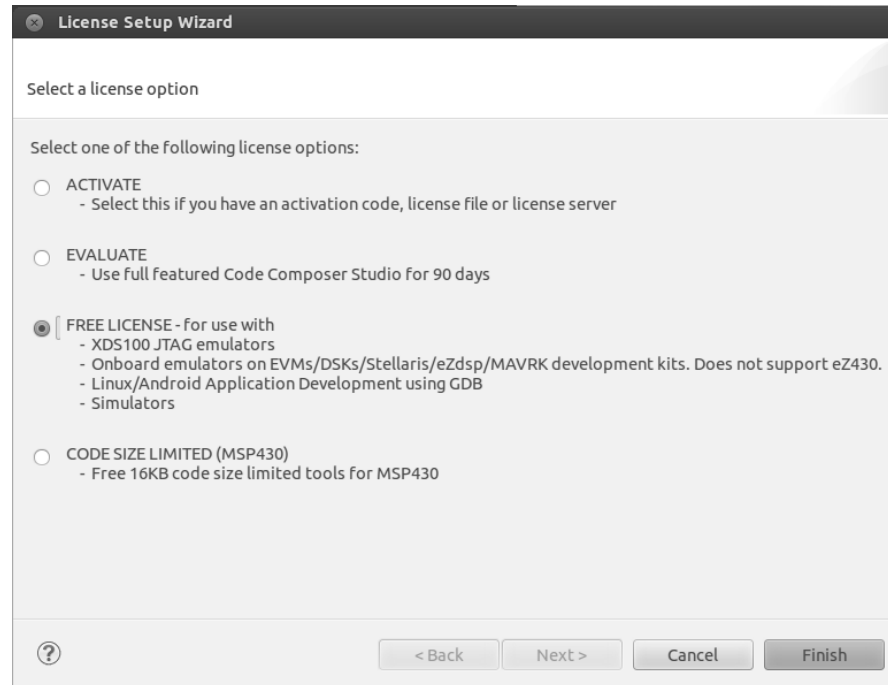
**22. Make the script executable**

```
# chmod a+x /home/user/ti/ccsv6/eclipse/my_ccstudio.sh
```

**23. Start Code Composer Studio**

**24. Select the Free License Agreement**

The License Agreement selection will come up upon first start.



If for some reason the License Setup Wizard does not automatically launch, you can access it via:

Help→Code Composer Studio Licenscing Information→Upgrade Tab→Launch Licensing Setup...

## 25. Close CCS

# Installing Workshop Files

The merged Yocto and Open Embedded projects provide an open-source set of recipe files for rebuilding various distributions using a tool named Bitbake. The generic Yocto/OE installation is one route to rebuilding Arago, but it is recommended that you instead install the Yocto/OE build environment from the [arago-project.org](http://arago-project.org) website as it has been pre-configured for Arago builds.

**1. Use aptitude to install the sg3-utils package (used in lab 0)**

```
ubuntu$ sudo apt-get install sg3-utils
```

**1. Use aptitude to install qvfb (used in lab 10)**

```
ubuntu# sudo apt-get install qt4-dev-tools
```

**1. Install bitbake dependencies**

```
ubuntu$ sudo apt-get install git build-essential diffstat texinfo  
gawk chrpath python-dev python-m2crypto
```

**2. Switch shell from dash to bash**

```
ubuntu$ sudo dpkg-reconfigure dash  
select "no" when prompted
```

**3. Install Linaro cross compiler**

```
ubuntu$ cd /home/user  
ubuntu$ wget --no-check-certificate \  
https://launchpad.net/linaro-toolchain-  
binaries/trunk/2013.03/+download/gcc-linaro-arm-linux-gnueabi-  
4.7-2013.03-20130313_linux.tar.bz2
```

```
ubuntu$ tar -jxvf gcc-linaro-arm-linux-gnueabi-4.7-2013.03-20130313_linux.tar.bz2 -C  
/home/user
```

**4. Add Linaro path to .bashrc**

```
ubuntu$ gedit /home/user/.bashrc
```

Add the following at the end:

```
export PATH=$PATH:/home/user/gcc-linaro-arm-linux-gnueabi-4.7-2013.03-  
20130313_linux/bin
```

**5. Source .bashrc for change to take effect**

```
ubuntu# source /home/user/.bashrc
```

**6. Install arago bitbake files**

```
Ubuntu$ git clone git://arago-project.org/git/projects/oe-layerssetup.git
```

**7. Configure for sdk-07.01.00.00**

```
Ubuntu$ cd oe-layerssetup  
Ubuntu$ ./oe-layertool-setup.sh -f configs/am sdk/am sdk-07.01.00.00-config.txt
```

**8. Add workshop overlay**

```
ubuntu$ cd /home/user/oe-layerssetup/sources  
ubuntu$ git clone https://github.com/preissig/meta-workshop
```

**9. Add meta-workshop layer to bitbake configuration**

```
ubuntu$ cd /home/user/oe-layerssetup/  
ubuntu$ gedit build/conf/bblayers.conf
```

Add "/home/user/oe-layerssetup/sources/meta-workshop" to the "BBLAYERS" variable.

**10. (Optional) Get preloaded packages**

Over time the locations and versions of packages change on the internet. Generally you will have the best success, especially with older builds, by downloading one of the preloaded source code packages. This is also significantly faster than having bitbake download each file individually.

```
Ubuntu$ wget
http://downloads.ti.com/dsps/dsps_public_sw/am_bu/sdk-
downloads/TISDK-Downloads/ALL/exports/amsdk-07.01.00.00-
downloads.tar.gz
Ubuntu$ tar xzf amsdk-07.01.00.00-downloads.tar.gz
Ubuntu$ mv sdk-7.1-downloads/* oe-layersetup/downloads
Ubuntu$ rmdir sdk-7.1-downloads
```

**11. Build sdk – this will take a long time, possibly 24 hours!**

```
Ubuntu$ cd build
Ubuntu$ source conf/setenv
Ubuntu$ MACHINE=am335x-evm bitbake -k arago-core-tisdk-image
(Note: the “-k” option will continue to build the distribution even if an error is encountered
in one of the packages.)
For available images:
# ls sources/meta-arago/meta-arago-distro/recipes-core/images/
```

**12. Copy sdk image**

```
ubuntu$ cp arago-tmp-external-linaro-toolchain/deploy/images/arago-core-tisdk-image-
am335x-evm.tar.gz /home/user
```

**13. De-archive sdk**

```
Ubuntu$ mkdir ti-sdk-07.01.00.00
Ubuntu$ tar xzf arago-core-tisdk-image-am335x-evm.tar.gz -C /home/user/ti-sdk-
07.01.00.00
```

**14. Run sdk installer**

```
ubuntu$ ./sdk-install.sh
```

**15. Run sdk set up**

```
ubuntu# sudo apt-get update
ubuntu$ sudo ./setup.sh
```

**When prompted, enter username as “user” instead of the default of “root.”**

**Accept default values for each other question until you get to the serial port for minicom. Instead of the default of “/dev/ttyS0” enter “/dev/ttyUSB1”**

**You may then accept the default values for the rest of the script.**

## Configuring Ubuntu Static IP

The “auto” setting for usb0 in /etc/network/interfaces is a workaround. It would be better specified as “allow-hotplug” however, there are known issues with this in Ubuntu 12.04. The web recommends using udev as an alternate solution, but the workshop developer was unable to make this approach work.

Using “auto usb0” works well, but with the disadvantage that if no ethernet-over-usb connection is available when Ubuntu starts up, the message “waiting on network configuration...” will appear and will require about 2 minutes to timeout. This extra 2 minutes of boot time may be circumvented by attaching the beaglebone so that the interface is present.

Users who dislike this 2 minute boot time may remove “auto usb0” in which case the usb0 will have to be manually configured each time the Beaglebone is attached using “#sudo ifup usb0”

**2. Open /etc/network/interfaces file**

```
# sudo gedit /etc/network/interfaces
```

**3. Add an “eth0” entry (or modify current entry.) Entry should be as follows:**

```
auto eth0
iface eth0 inet static
    address 192.168.1.1
    netmask 255.255.255.0
```

```
auto usb0
iface usb0 inet static
    address 192.168.7.1
    netmask 255.255.255.0
```

Note: “address” and “netmask” entries preceded by tab.

**4. Save and close**

**5. Remove Gnome networking settings**

```
# sudo nm-connection-editor
```

Any connection that appears under the “wired” or “wireless” tab should be deleted.

**6. (Optional) Reboot and use “ifconfig” to verify new setting**

```
# ifconfig
```

**7. Open /etc/hosts**

```
# sudo gedit /etc/hosts
```

**8. Add static IP addresses for hosts on the network**

(At the end of the file, add the following)

```
192.168.1.1 ubuntu.gigether.net
192.168.1.2 am335x.gigether.net
192.168.7.1 ubuntu.etherusb.net
192.168.7.2 am335x.etherusb.net
```

## (Optional) Installing Gnome3 and Standard Scrollbars

Ubuntu 12.04 ships with a desktop manager called Unity. One feature that a lot of people do not prefer in Unity is that the drop-down lists that would normally appear at the top of a window (including CCS) now appear at the top of the desktop. Additionally, Unity uses a new type of scrollbar called overlay scrollbars that, while saving a little space on the screen that can be used for other things, are a little more difficult to use.

This section is not required for the workshop labs to work properly, but since these changes were made on the workshop image, they are listed here.

1. **Launch a terminal**
2. **Acquire a WAN (i.e. internet) connection**  
If you have already set up a static IP address as per the previous section, you can override the static address using

```
ubuntu$ sudo ifdown eth0
ubuntu$ sudo dhclient eth0
```
3. **Install gnome-shell Aptitude package**

```
ubuntu$ sudo add-apt-repository ppa:gnome3-team/gnome3
ubuntu$ sudo apt-get update
ubuntu$ sudo apt-get install gnome-shell
```
4. **Log out of the Ubuntu session**  
There is a gear icon in the top right corner that produces a drop-down menu with the logout option.
5. **Select the Gnome Desktop**  
Click the Ubuntu icon next to the username (user) and select Gnome.
6. **Press login to log back in**  
The desktop has only subtly changed, but if you launch CCS, you will notice that the pulldown menus are now at the top of the CCS window (instead of along the top of the desktop.)
7. **Disable overlay scrollbars**  
Launch a terminal and type the following (single line, no carriage return)

```
ubuntu$ gsettings set org.gnome.desktop.interface ubuntu-overlay-scrollbars false
```
8. **Log out and back in for change to take effect**  
The “user” dropdown menu in the top right of the desktop can be used to log out.