Texas Instruments

Introduction to Embedded Linux

Student Guide

Revision 1.01
January 2014

TTO

Technical Training Organization

Important Notice

Revision History

August 2013 – Revision 1.0

Mailing Address

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

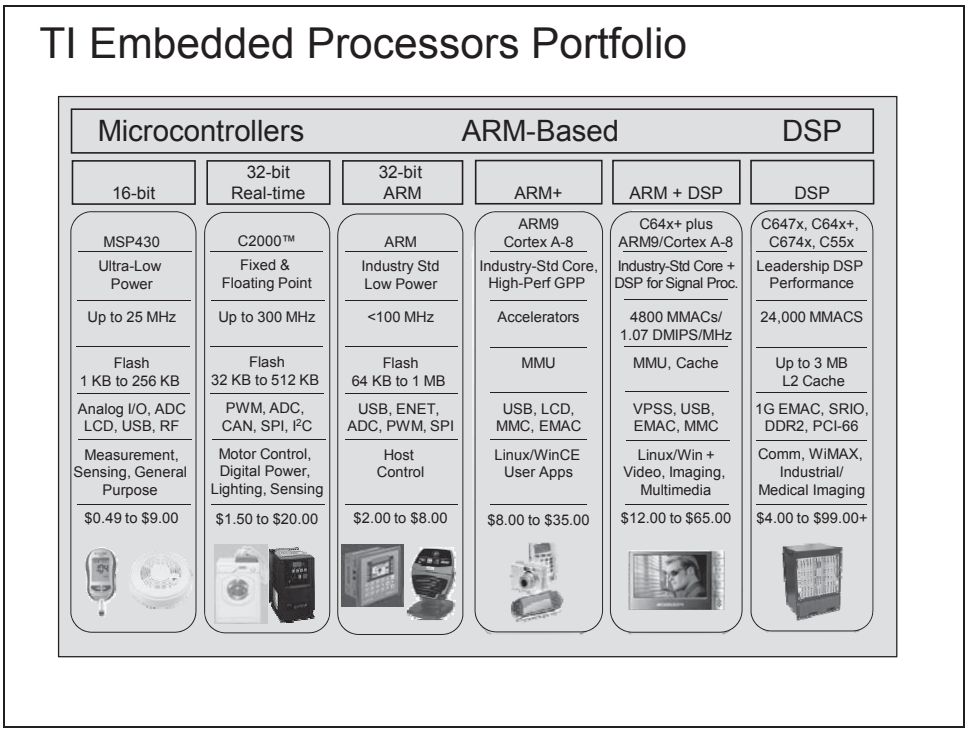
Module 01: Booting Linux

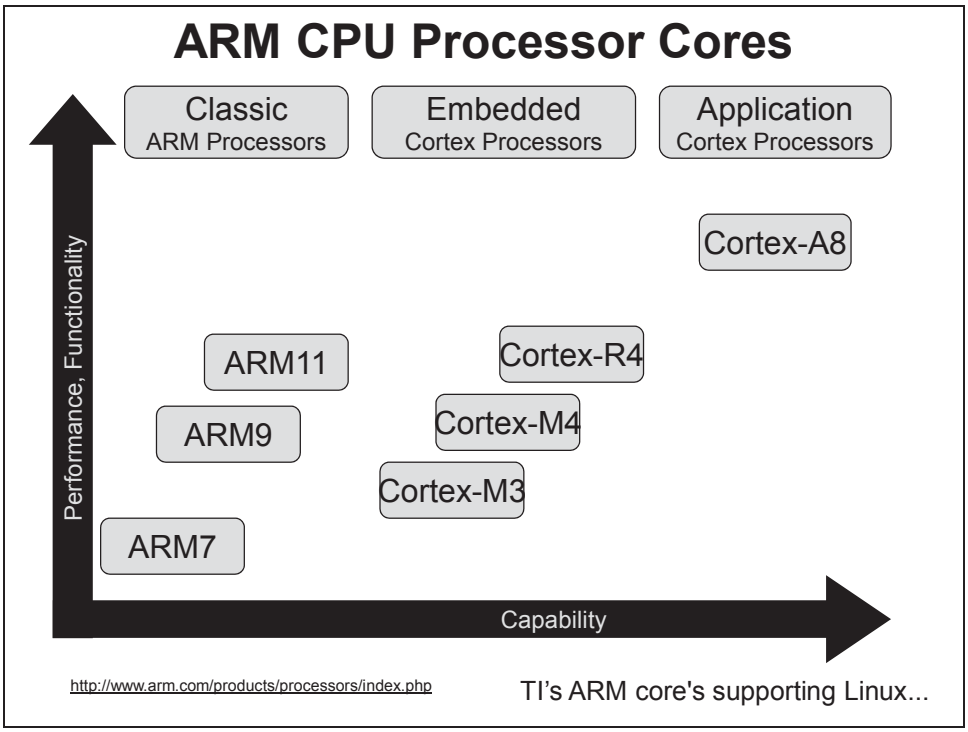
Introduction

This module begins by showing how a Linux distribution may be booted using a provided u-boot bootloader, Linux kernel and Linux filesystem. Next it shows how those same three elements may be rebuilt from source code and possibly reconfigured. Finally, the module covers some details of the “System five” (sys-V) initialization standard that is used to specify the startup configuration of many Linux distributions, including Arago (Beaglebone distribution) and Ubuntu (x86 Distribution.)

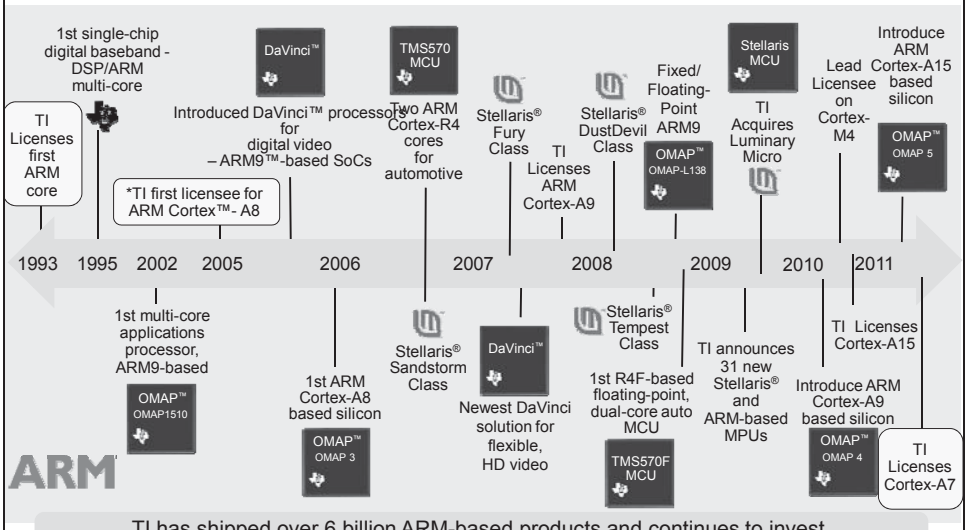
Module Topics

(Short) Product Overview





TI ARM® investment and innovation



TI has shipped over 6 billion ARM-based products and continues to invest in a large portfolio of scalable platforms from $1 to >1GHz

* TI licensed in July 2003, but publicly announced Oct 2005.

AM335x Cortex™-A8 based processors

Benefits
- High performance Cortex-A8 at ARM9/11 prices
- Rich peripheral integration reduces complexity and cost

Sample Applications

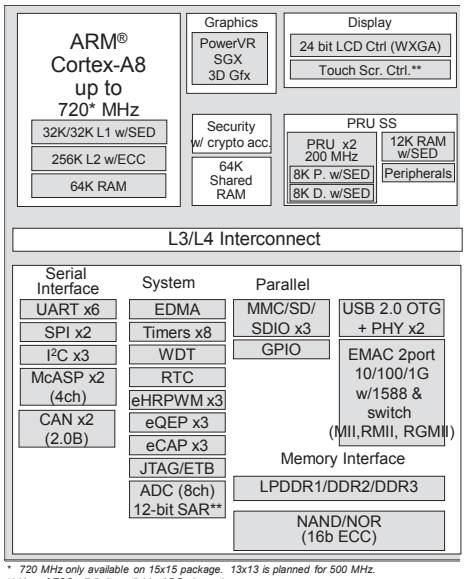
- Home automation
- Home networking
- Gaming peripherals
- Consumer medical appliances
- Printers
- Building automation
- Smart toll systems
- Weighing scales
- Educational consoles
- Advanced toys
- Customer premise equipment
- Connected vending machines

Software and development tools
- Linux, Android, WinCE and drivers direct from TI
- StarterWare enables quick and simple programming of and migration among TI embedded processors
- RTOS (QNX, Wind River, Mentor, etc) from partners
- Full featured and low cost development board options

Power Estimates
- Total Power: 600mW-1000mW
- Standby Power: ~25mW
- Deep Sleep Power: ~5-7mW

Schedule and packaging
- Samples: October 31, 2011; Production: 2Q'12
- Dev. Tools: Order open October 31, 2011
- Prelim docs: available today
- Packaging: 13x13, 0.65mm via channel array
  15x15, 0.8mm

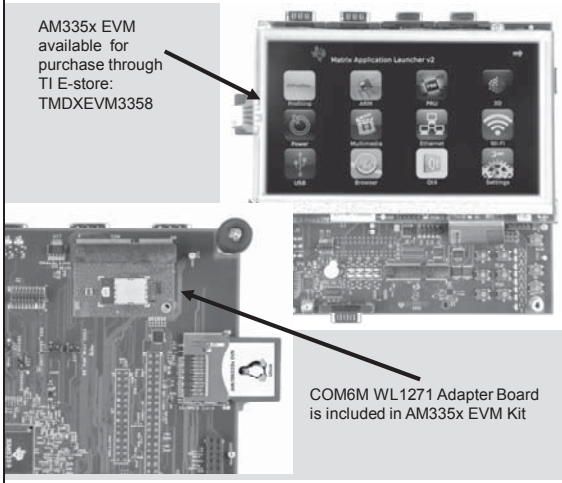
Availability of some features, derivatives, or packages may be delayed from initial silicon availability
Peripheral limitations may apply among different packages
Some features may require third party support
All speeds shown are for commercial temperature range only



** 720 MHz only available on 15x15 package. 13x13 is planned for 500 MHz.*
*** Use of TSC will limit available ADC channels.*
SED: single error detection/parity

AM335x/WL1271 Development Kit

AM335x EVM available for purchase through TI E-store: TMDXEVM3358



COM6M WL1271 Adapter Board is included in AM335x EVM Kit

Kit Contents

- TI AM335x EVM Kit includes AM3358 Microprocessor, 512 MB DDR2, TPS65910 Power management IC, 7" LCD, Software & Tools, WL1271 COM6M Adapter Card

Demos

- QT based WLAN and Bluetooth® demo Applications integrated into SDK
- WPA Supplicant and Host Apd GUIs for WiFi Station and Soft AP setup
- Profusion Bluetooh GUI with BMG (Scan, Pair, Connect), A2DP, FTP, OPP and SPP demos

Software

- Open Source Linux 3.2 drivers
- Pre-Integrated with TI Linux SDK
- mac80211 Open Source WLAN Driver
- BlueZ Open Source Bluetooth® stack
- BlueZ OpenObex Profiles

Documentation

- User and Demo guide, Releases: TI Wireless Connectivity Wiki
- Module WL1271-Type TN Datasheet from Murata

WL1271-TypeTN (Murata)

WLAN 802.11 b/g/n and Bluetooth® v4.0 BLE Module

LBEE5ZSTNC-523

Features

- IEEE 802.11 b/g/n compliant
- Bluetooth 4.0 with Bluetooth Low Energy
- Wi-Fi Direct
- TI's proven 6th generation Wi-Fi and Bluetooth solution
- Pre-integration with high performance Cortex-A8 based AM335x processor platform
- Open-source compliant Wi-Fi and Bluetooth drivers
- FCC Certified, ETSI & EMC Tested WL1271 module
- Sample applications and demos

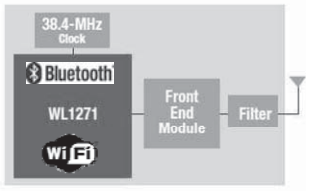
Applications

- Mobile consumer devices
- Industrial and home automation, metering
- Portable data terminals
- Video conferencing, video camera

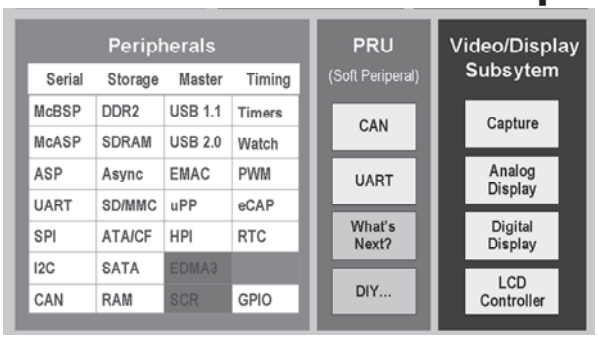
www.ti.com/wl1271typetn

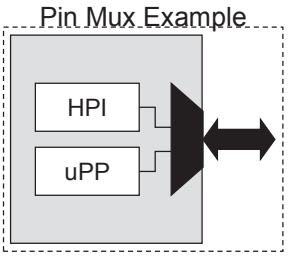
Benefits

- Seamless, direct and high throughput Wi-Fi connectivity between devices (no external access points needed)
- High throughput, reliable signal integrity, best in class coexistence, enhanced low power
- Simplified and reduced hardware and software integration effort, get started quickly
- Platform enables high performance processing and increased level of integration at value-line pricing
- Open-source compliant Wi-Fi and Bluetooth drivers
- Certified modules lowers manufacturing and operating costs, saves board space and minimizes RF expertise required



What is Pin Multiplexing?

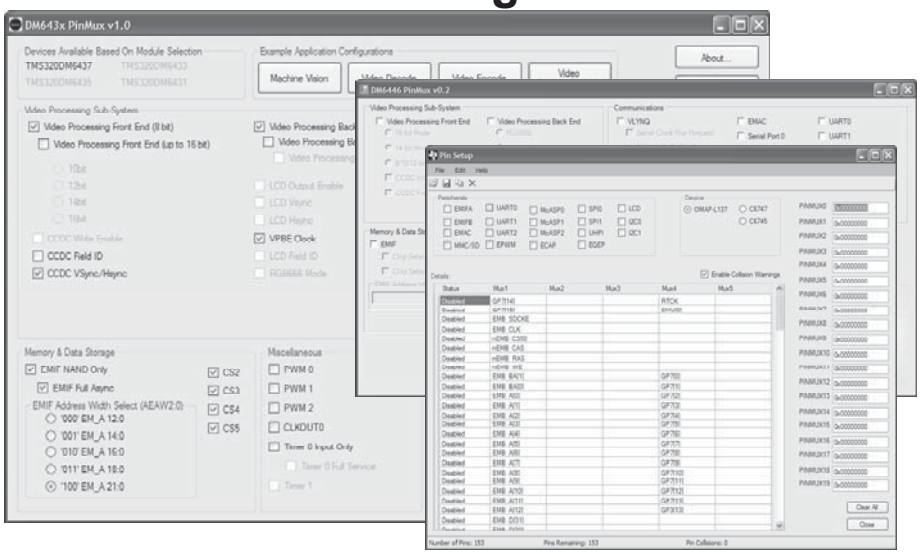




- How many pins is on your device?
- How many pins would all your peripheral require?
- Pin Multiplexing is the answer – only so many peripherals can be used at the same time … in other words, to reduce costs, peripherals must share available pins
- Which ones can you use simultaneously?
  - Designers examine app use cases when deciding best muxing layout
  - Read datasheet for final authority on how pins are muxed
  - Graphical utility can assist with figuring out pin-muxing…

Pin mux utility...

Pin Muxing Tools



- Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- Provides Pin Mux Register Configurations

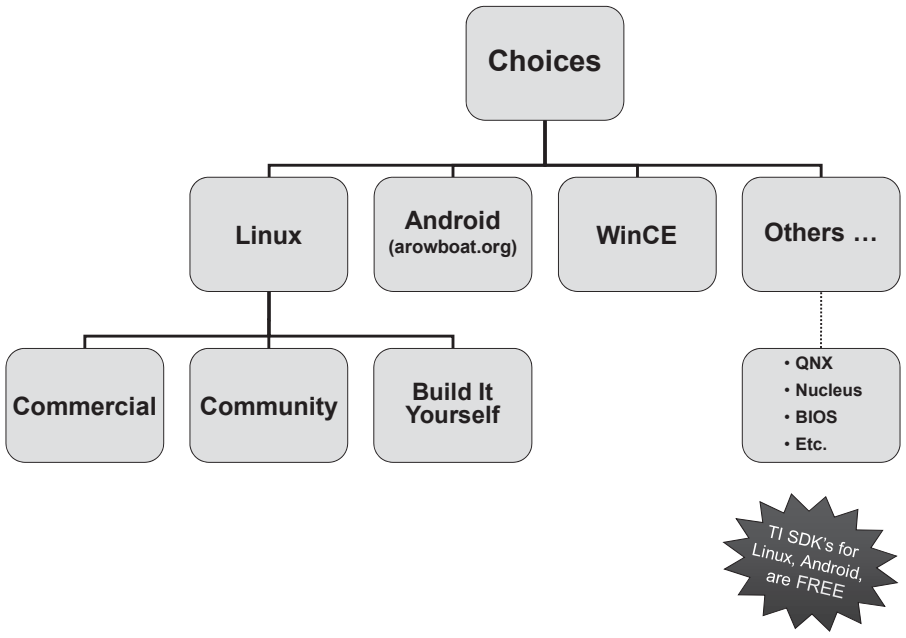
Linux Distributions

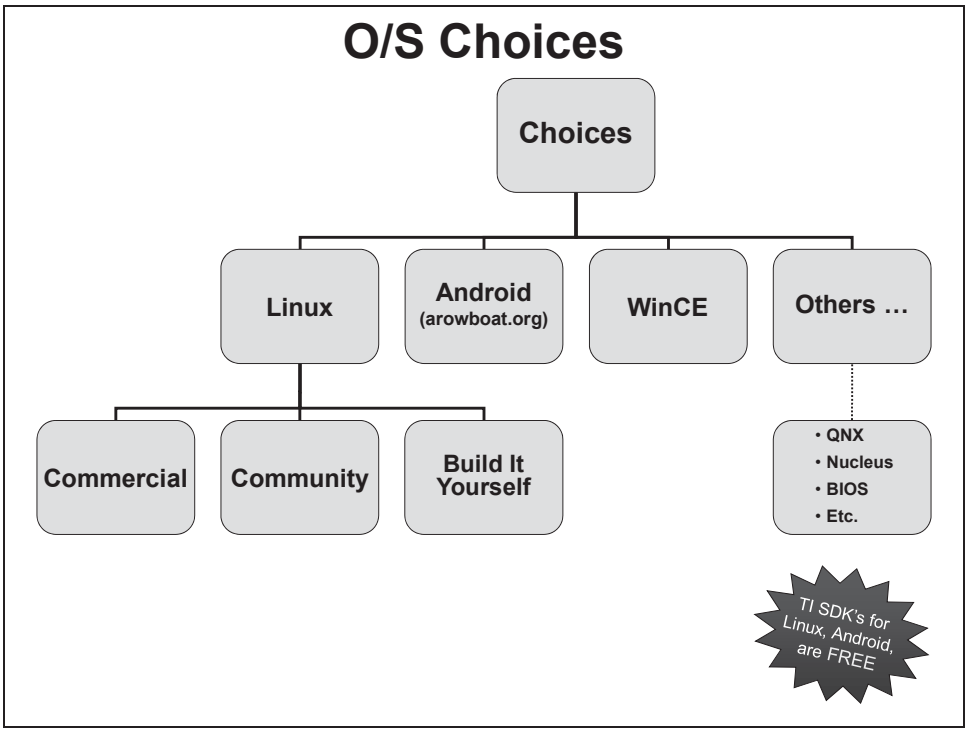
What Is a 'Linux Distribution'

A 'Linux distribution' is a combination of the components required to provide a working Linux environment for a particular platform:

1. Linux kernel port
   - A TI LSP or Linux PSP is a Linux kernel port to a device, not just a set of device drivers
2. Bootloader
   - Uboot is the standard bootloader for ARM Linux
3. Linux 'file system'
   - This does NOT mean a specific type of file system like FAT file system or flash file system … rather, it more like the "C:\" drive in Windows
   - It refers to all the 'user mode' software that an application needs such as graphics libraries, network applications, C run-time library (glibc, uclibc), codec engine, dynamically-loaded kernel modules (CMEM, DSPLINK)
4. Development tools
   - Code Composer Studio
   - Angstrom cross-gcc

O/S Choices



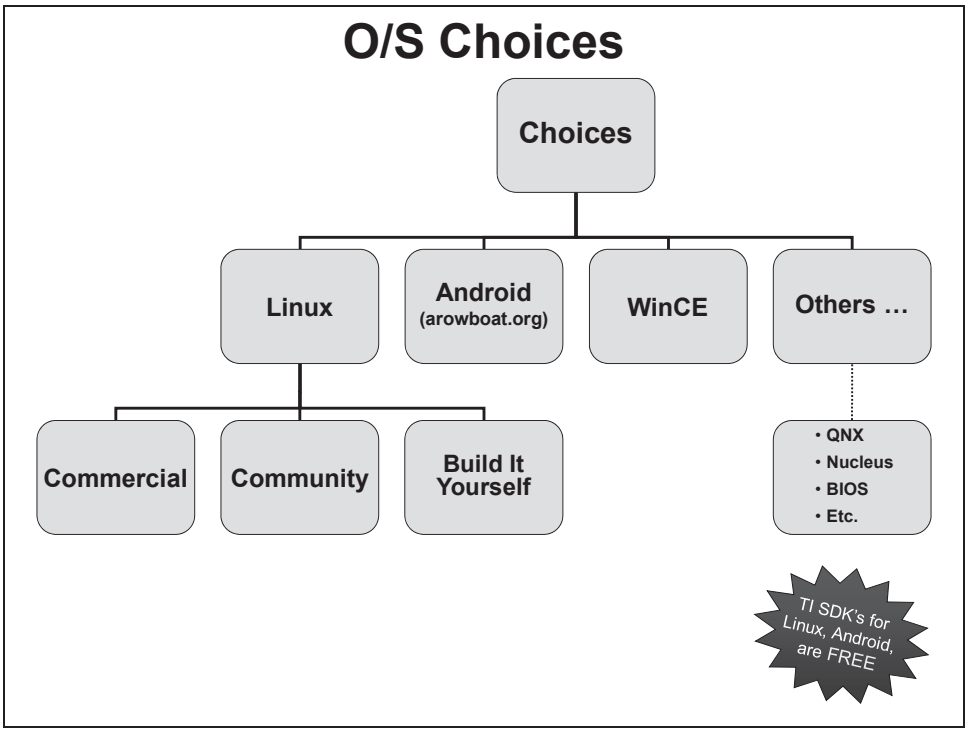


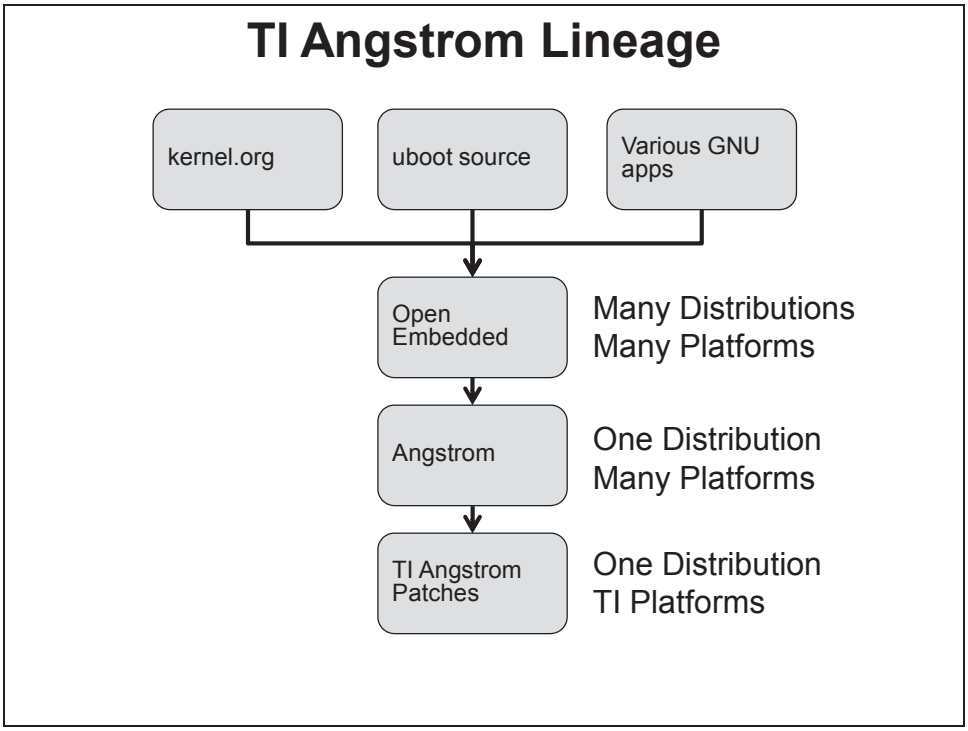
Commercial O/S Vendors

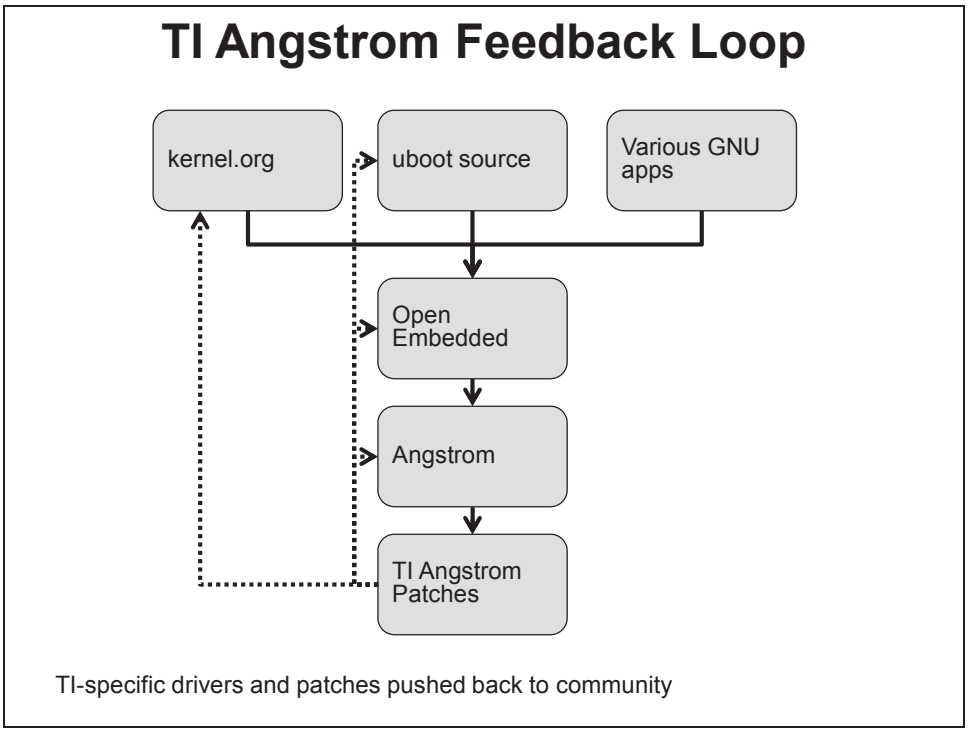
- Linux
  - MontaVista
  - TimeSys
  - Wind River
  - Mentor
  - Ridgerun
- WinCE
  - Adeneo
  - Mistral
  - MPC Data
  - BSQUARE
- RTOS
  - Green Hills
  - Wind River (VxWorks)
  - ELogic (ThreadX)
  - QNX
  - Mentor (Nucleus)

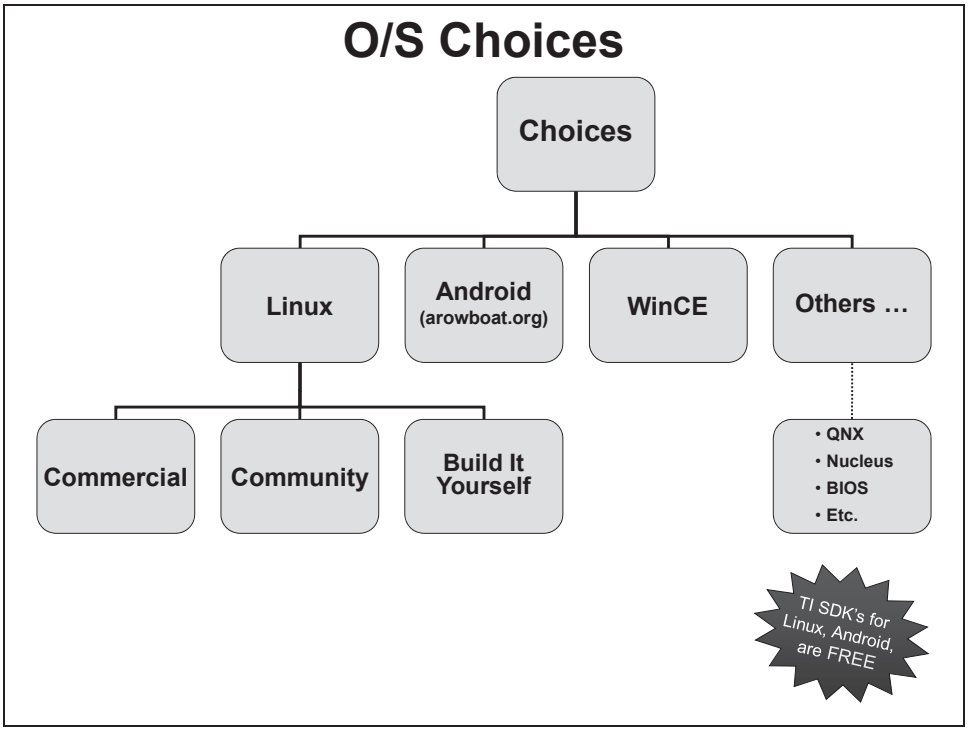
Linux Partner Strategy

- Commercial: provide support, off-the-shelf Linux distributions or GNU tools
- Consultants: provide training, general embedded Linux development expertise, or specific expertise for developing drivers or particular embedded applications


O/S Choices
Choices
Linux
Android
(arowboat.org)
WinCE
Others …
Commercial
Community
Build It
Yourself
• QNX
• Nucleus
• BIOS
• Etc.
TI SDK's for
Linux, Android,
are FREE


TI Angstrom Lineage
kernel.org
uboot source
Various GNU
apps
Open
Embedded
Many Distributions
Many Platforms
Angstrom
One Distribution
Many Platforms
TI Angstrom
Patches
One Distribution
TI Platforms


TI Angstrom Feedback Loop
kernel.org
uboot source
Various GNU apps
Open Embedded
Angstrom
TI Angstrom Patches
TI-specific drivers and patches pushed back to community


O/S Choices
Choices
Linux
Android (arowboat.org)
WinCE
Others …
Commercial
Community
Build It Yourself
• QNX
• Nucleus
• BIOS
• Etc.
TI SDK's for Linux, Android, are FREE

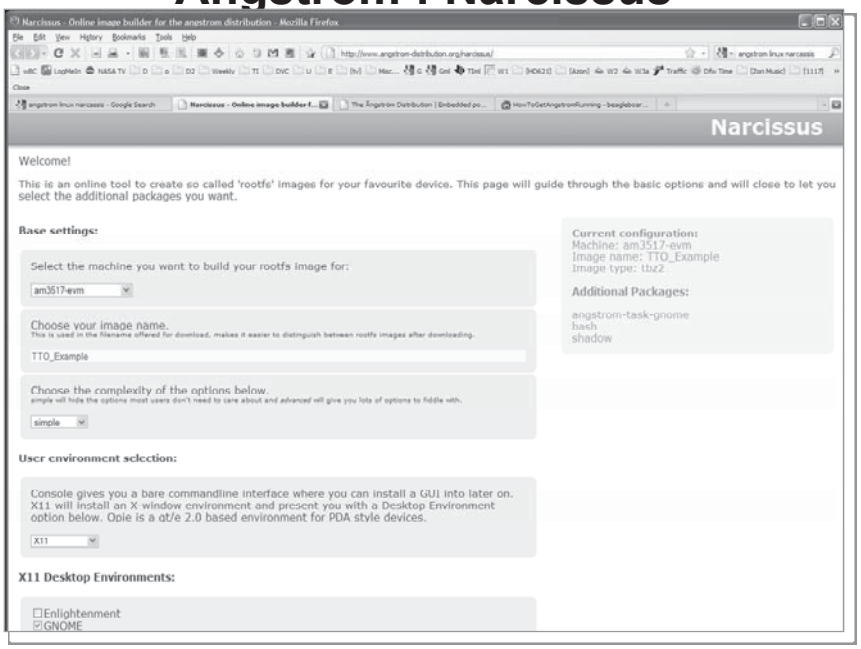
Bitbake Recipe File

```
aDESCRIPTION = "nodeJS Evented I/O for V8 JavaScript"
HOMEPAGE = "http://nodejs.org"
LICENSE = "MIT"
DEPENDS = "openssl"
SRC_URI = "  http://nodejs.org/dist/node-v${PV}.tar.gz
       file://libev-cross-cc.patch
       file://node-cross-cc.patch "
SRC_URI[md5sum] = "c6051dd216817bf0f95bea80c42cf262 "
S = "${WORKDIR}/node-v${PV}"
do_configure () {
       ./configure --prefix=${prefix} --without-snapshot
}
do_compile () { make }
do_install () { DESTDIR=${D} oe_runmake install }
```

http://www.openembedded.org/wiki/How_to_create_a_bitbake_recipe_for_dummies

Includes download instructions, build instructions and dependencies

Ångström : Narcissus



(http://www.angstrom-distribution.org/narcissus)

Using opkg to Add New Packages

To see the commands the opkg package manager supports

```
host$ opkg
```

To install an opkg-compliant package

```
host$ opkg install <packagename>.ipkg
```

To download a package but not install it

```
host$ opkg download <packagename>
```

To find out which packages provides a given file

```
host$ opkg whatprovides <filename>
```

To list which packages are available

```
host$ opkg list
```

To update the list of available packages and “whatprovides” search

```
host$ opkg update
```

Linux Distributions Options for TI

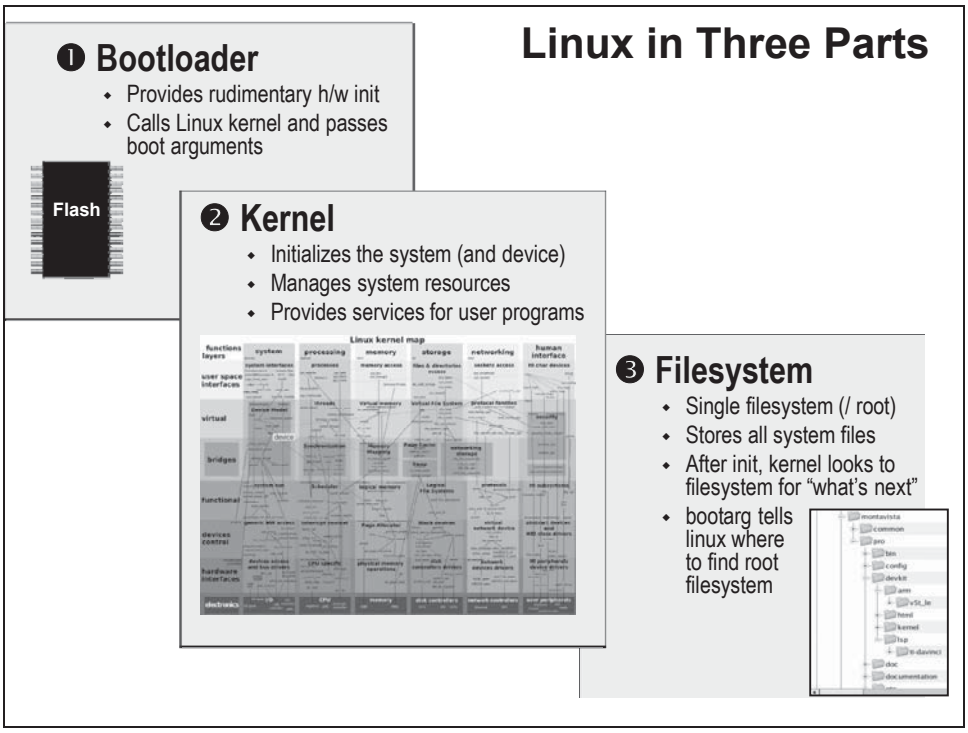
Commercial	Community		Custom (Build it Yourself)	
• **Timesys** • **MontaVista** • **Etc.**	**TI SDK (PSP)**	**Ångström / Arago**	**Open Embedded (OE)**	**Custom from Sources**
• Binary (Update patches)	• Binary Updated for each SDK release	• Binary • Narcissus (online tool) • opkg	• Bit-Bake • Recipies	• “GIT” from kernel.org, and others

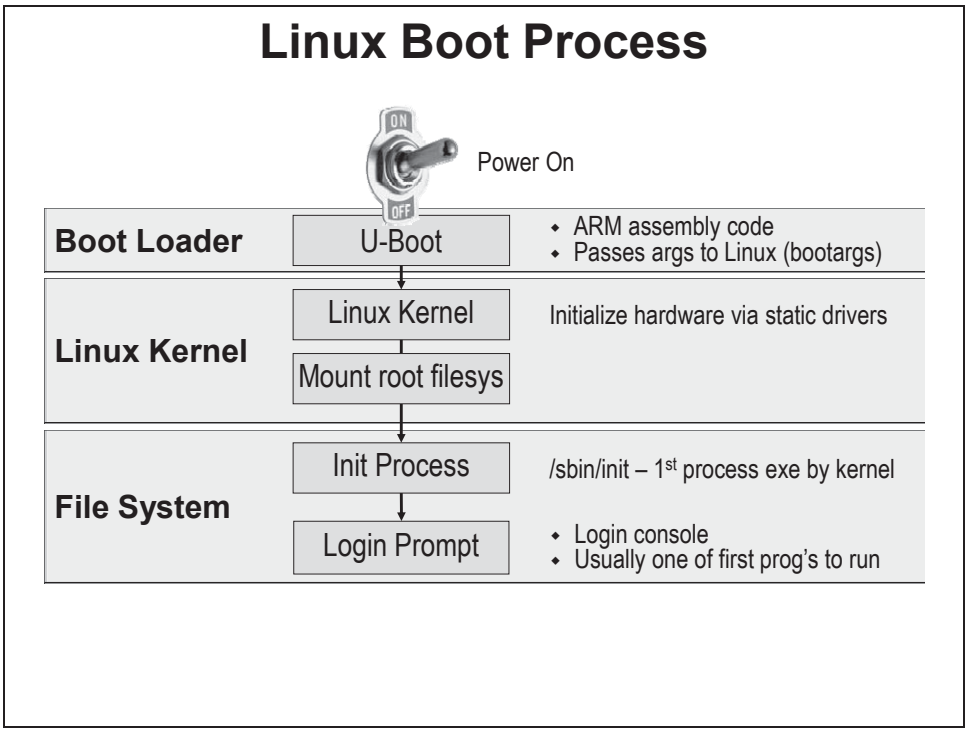
Ease of Use

• Easy
• Tested

Experienced User •
Latest •

Booting Linux from Pre-Built Binaries





Where Do You Find …

Where located:	Flash Boot		
1a. SBL, UBL or MLO	n/a		
1b. Bootloader (U-Boot)	Flash		
2. Linux Kernel	Flash		
3. Filesystem	Flash		

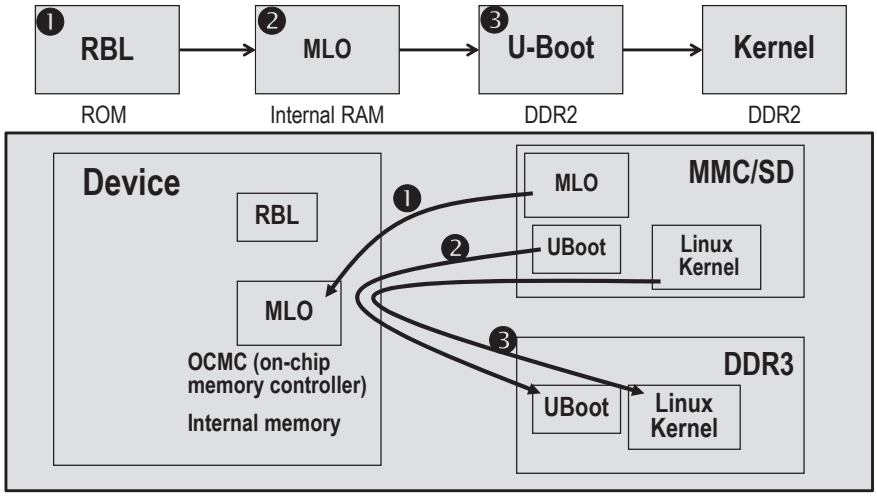
- ROM bootloader supports NAND, NOR or SPI-flash boot (determined by BOOTM pins high/low at device powerup)
- Linux supports flash-based filesystems such as YAFFS and JFFS
- All components may be flashed using U-boot (initial load via MMC or bootp) or JTAG using flashing utility via Code Composer Studio

Where Do You Find …

Where located:	Flash Boot	MMC Boot	
1a. SBL, UBL or MLO	n/a	MMC	
1b. Bootloader (U-Boot)	Flash	MMC	
2. Linux Kernel	Flash	MMC	
3. Filesystem	Flash	MMC	

- Multimedia Card (MMC) or Secure Digital card (SD) may be flashed using an MMC/SD programmer using a variety of utilities
- Simple, low cost method for booting Linux (or just U-boot) on a development board that has nothing pre-programmed (or for recovery.)

Booting Linux – MMC/SD Boot



The ROM bootloader cannot make assumptions about external memory, so it can only load to the device's internal memory

uBoot with MMC driver is too large to fit into internal memory

Where Do You Find …

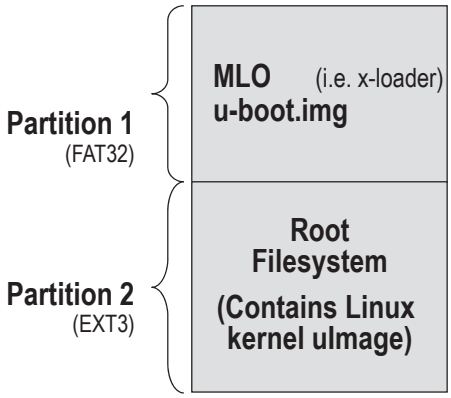
Where located:	Flash Boot	MMC Boot	NFS Boot
1a. SBL, UBL or MLO	n/a	MMC	bootp / tftp
1b. Bootloader (U-Boot)	Flash	MMC	tftp
2. Linux Kernel	Flash	MMC	tftp
3. Filesystem	Flash	MMC	nfs

- NFS boot is typically used for development but not production devices
- All components of the system are loaded from host server at each boot
- Filesystem changes on host are instantly reflected
- UBL, U-boot and Linux Kernel changes are reflected on each reboot
- Good method to ensure uniformity across multiple development boards

Boot Linux from MMC

Attach SD/MMC programmer with inserted Micro-SD adapter

```
host $./mkcard.txt
host $ cp MLO /media/boot
host $ cp u-boot.img /media/boot
host $ tar -zxf Angstrom.tar.gz -C /media/Angstrom
```



ROM bootloader will load a file named "MLO" from partition 1, which must be FAT32 filesystem

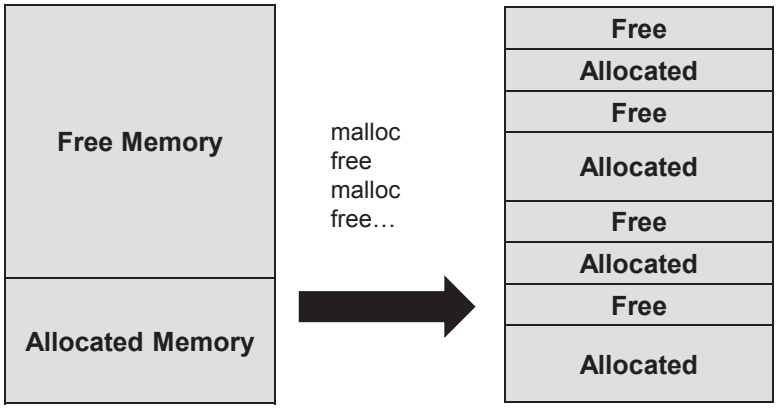
MLO (per MMC default config) boots a file named "u-boot.bin" from partition 1 of FAT32 MMC

Linux root filesystem must be in its own partition.

MMU and Dynamic Libraries

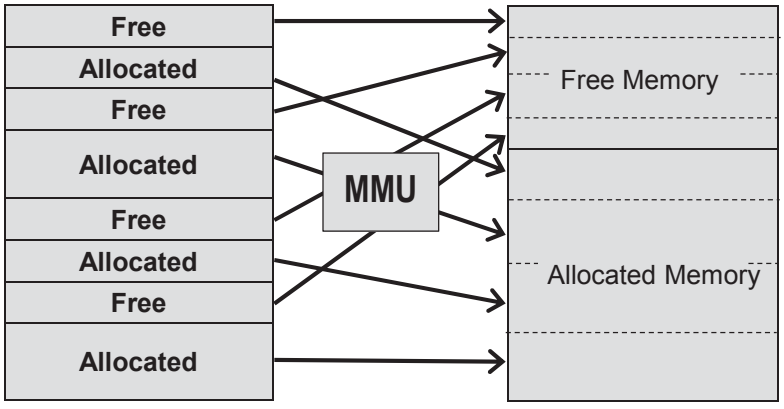
Fragmentation

Problem:
Over time, the system heap that is used to manage dynamic memory becomes fragmented.



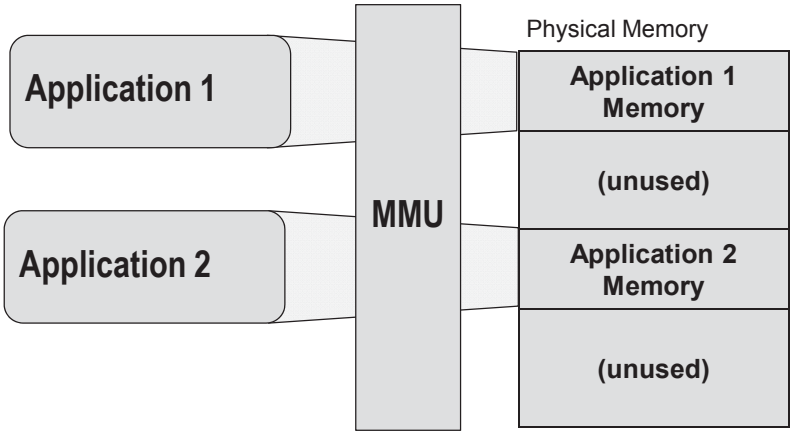
Virtual Memory

Solution:
A Memory Management Unit (MMU) maps fragmented physical memory into "virtually contiguous" memory



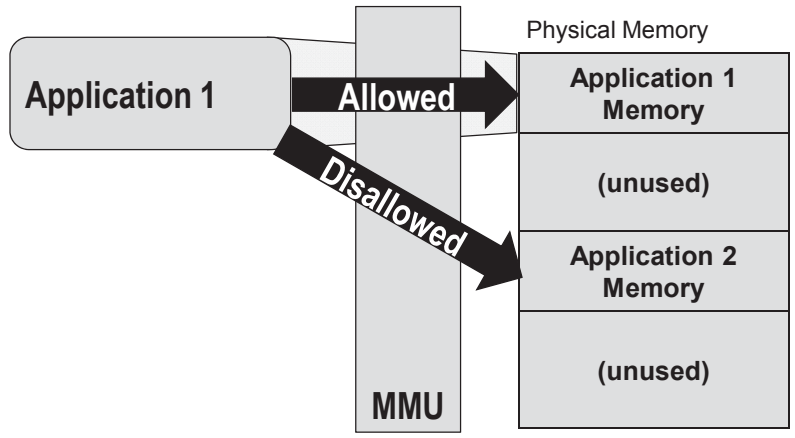
Memory Protection

The memory management unit also provides memory protection by denying access to regions of physical memory that are not allocated to a given application

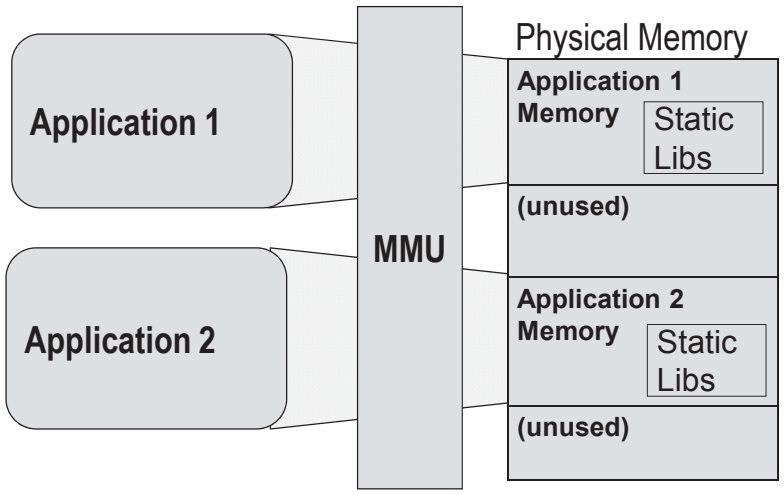


Memory Protection

When application 1 is executing, it's application memory is mapped to a valid virtual address. Disallowed memory is assigned no virtual address and is therefore unreachable.

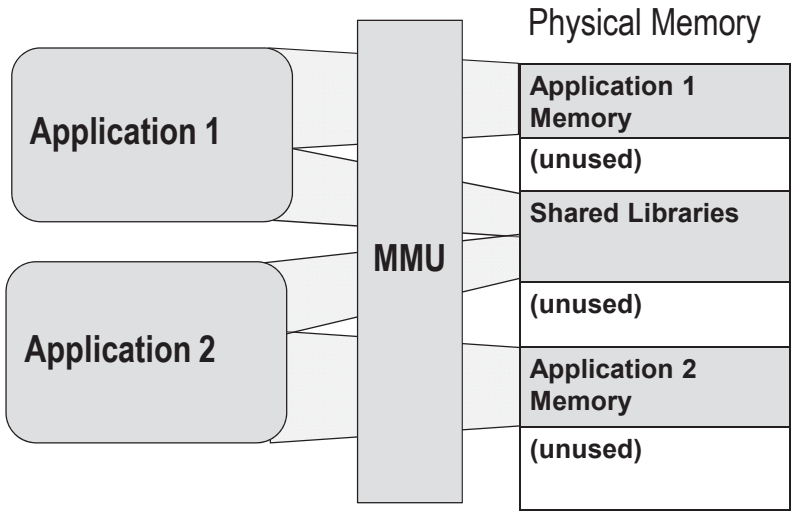


Static Linking and .a Libraries (Archives)



Libraries are shared across applications and dynamically linked

Dynamic Linking and .so Libraries



Common libraries are shared across applications and dynamically

Linking Linux Libraries

```
# gcc myfile.c –lm -lpthread
```

This links the dynamic libraries:
- libm.so – shared math library
- libpthread.so – shared POSIX thread library

or the static libraries (archives)
- libm.a – static math library
- libpthread.a – static POSIX thread library

Shard object libraries link by default, unless overridden with the option:
- --static

(Page intentionally left blank)

Lab 1: Booting Linux

Introduction

This lab exercise will demonstrate creating a bootable mini-SD card and using it to boot the Angstrom Linux distribution on the Beaglebone Black. 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

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. **Attempt to boot the Beaglebone Black without a micro-SD card. Be sure the USB cable is attached between Beaglebone and the host PC.**

   The Beaglebone Black will boot from eMMC (embedded MMC) on the board if possible. Before booting from the micro-SD card you need to verify that the eMMC boot has been disabled.
3. **(if necessary) Remove the MLO from the Beaglebone Black eMMC**

   If the Beaglebone Black boots successfully, user LEDs near the USB connection will show activity, including a heartbeat on LED0. If this occurs, the first partition of the eMMC will appear as an external drive on the host PC. The Beaglebone exposes this partition using a mass storage gadget driver over USB. If this occurs, erase the MLO file from this partition.

   If you are using a workshop-supplied board, it is likely that this has been done already and this step will not be necessary.

   If you are using your own board out of the box, this step will need to be done.
4. **(if necessary) Repeat step 2 to verify Beaglebone will not boot from eMMC**
5. **Insert the micro-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.

   Don’t worry; we will reformat the card in step 8 to ensure that the lab exercise is completed successfully.
6. **Launch an xterm terminal.**

   ctrl-alt-t

7. **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
   ```

8. **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 7 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 these partitions and it may or may not have them mounted. If there is no partition, or it is not mounted, you will get a warning when you attempt to unmount, but this can be ignored.

9. **Change into the lab01 directory**

   ```
   ubuntu$ cd /home/user/labs/lab01_boot_linux
   ```

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

    ```
    ubuntu$ ls
    ```

    You should see two gnu-zipped (gz) tape archive (tar) files – boot.tar.gz and rootfs.tar.gz – and also a script file "create-sdcard.sh"

    The MLO and u-boot files are archived into "boot.tar.gz"

    The rootfs.tar.gz contains the filesystem, as well as the kernel uImage in the /boot directory.

11. **Test the contents of the "boot.tar.gz" archive.**

    It is a good idea before expanding an archive to test its contents to see what will be written. Let's verify that the archive "boot.tar.gz" has the boot files we expect to place in partition 1 of the multimedia card.

    ```
    ubuntu$ tar ztf boot.tar.gz
    ```

12. **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.

13. **(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 7. There should be only one option, so enter "1" and continue.

14. **(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.

15. **(create-sdcard.sh) Enter "y" to continue after partitioning is complete.**
16. **(create-sdcard.sh) Choose option "2" to use custom files for the boot partition and root file system.**

The files for the root partition are actually exactly the same as ship with the SDK; however, a few minor changes have been made to the file system, so you will need to install the custom file system.

17. **(create-sdcard.sh) Specify the "boot.tar.gz" archive to be copied into partition 1.**

```
Enter the path for Boot Partition: boot.tar.gz
```

Specify "boot.tar.gz" as the archive of files to be placed in the boot partition.

18. **(create-sdcard.sh) Specify the "rootfs.tar.gz" archive to be copied into partition 2.**

```
Enter the path for Rootfs Partition: rootfs.tar.gz
```

Specify "rootfs.tar.gz" as the archive of files to be placed in the root file system partition.

19. **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.

20. **Remove the mini-SD card from the SD/MMC card reader and insert it into the mini-SD slot of Beaglebone Black.**

On the Beaglebone Black, the metal leads of the micro-SD card should face away from the printed circuit board. Note that the micro-SD slot is spring loaded.

B. Boot Linux on the Beaglebone Black

21. **Connect an Ethernet cable between the host PC and the Beaglebone Black.**
22. **Connect a USB cable between the host PC and the Beaglebone Black.**
    Once this connection is made, it will supply power to the Beaglebone Black, which will begin to boot. Upon a successful boot, you will see activity on the blue LEDs next to the USB connector on the Beaglebone, including a heartbeat on LED0.
23. **Verify IP addressing**

    The networking connections have been set with static addresses.

    To verify the address of the gigabit Ethernet connection and Ethernet-over-USB:

    ```
    ubuntu$ ifconfig

    eth0:      192.168.1.1

    lo:        127.0.0.1

    usb0:      192.168.2.1
    ```

    (lo is the local loopback connection.)

    To find the address of the Beaglebone Black on the gigabit Ethernet connection:

    (type the following into the terminal without a carriage return)

    ```
    ubuntu$ for i in {1..254}; do ping -c 1 192.168.1.$i
                -w 2 | grep from; done

    64 bytes from 192.168.1.1: icmp_req=1 ttl=64 time=0.035 ms

    64 bytes from 192.168.1.2: icmp_req=1 ttl=64 time=0.206 ms
    ```

    (note: the script will try all 254 addresses on the subnet, so if you don't want to wait for 252x2 second timeouts, press ctrl-C to exit)

    Since the Ubuntu address is 192.168.1.1, we know the Beaglebone Black must be at 192.168.1.2
24. **View the /etc/hosts file**

    ```
    ubuntu$ cat /etc/hosts
    ```

    Note that hostnames have been assigned to the static addresses used in this workshop.

    ```
    192.168.1.1      ubuntu.gigether.net

    192.168.1.2      beaglebone.gigether.net

    192.168.2.1      ubuntu.etherusb.net

    192.168.2.2      beaglebone.etherusb.net
    ```

25. Establish a secure shell (ssh) connection between host and Beaglebone

```
ubuntu$ ssh root@beaglebone.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 beaglebone.gigether.net (192.168.1.2)' can't be established…`"

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

If you get an error message when attempting to connect ssh, execute the "clear_ssh.sh" script from the "lab00_utils" directory:

```
ubuntu$ ~/labs/lab00_utils/clear_ssh.sh
```

This script will clear any key that may currently exist for the Beaglebone, allowing the generation of a new key via the procedure above.

26. Press "Enter" key when prompted for password

There is no root password on the Beaglebone Black, so when ssh prompts you for a password, simply press "Enter."

C. Basic Terminal Manipulation

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

   ```
   beaglebone$ cd ~
   ```

   The tilde "~" character is Linux shorthand for the current user's home directory, which is located at "/home/root"

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

   ```
   beaglebone$ pwd
   ```

   The "pwd" command is short for "print working directory."

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

   ```
   beaglebone$ ls
   ```

4. **Make a new directory named "lab01."**

   ```
   beaglebone$ mkdir lab01
   ```

5. **Change into the "lab01" directory.**

   ```
   beaglebone$ cd lab01
   ```

6. **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.

   ```
   beaglebone$ touch myhugelongfilename1
   ```

7. **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.

   ```
   beaglebone$ ls m<tab>
   ```

8. **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 7, and the second press will give you the touch command of step 6. 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.

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

```
beaglebone$ ls m<tab>
```

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

```
beaglebone$ 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.

```
beaglebone$ ls myhugelongfilename<tab><tab>
```

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

```
beaglebone$ ls myhugelongfilename
```

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

10. **Test the echo command**
The echo command redirects the supplied text to standard output (i.e. the terminal.)

```
beaglebone$ echo Beaglebone is a great Linux platform!
```

11. **Add to “myhugefilename1” by redirecting the echo command into the file.**

```
beaglebone$ echo Beaglebone rocks! > 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.

Note that a single greater than (“>”) will overwrite the entire file, while double greater than (“>>”) will append to the end of the file.

12. **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.)

```
beaglebone$ cat myhugefilename1
```

13. **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.

```
beaglebone$ more myhugefilename1
```

14. **Create a subdirectory named "subdir."**

    Refer to step 4 if needed

15. **Create a copy of "myhugefilename1" named "mycopy1," where the file "mycopy1" is in the "subdir" directory.**

```
beaglebone$ cp myhugefilename1 subdir/mycopy1
```

16. **Create a second copy using the current directory reference "."**

```
beaglebone$ 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.

17. **Verify the copies by listing the contents of the subdir directory.**
18. **Change into the subdir directory.**
19. **Change the name of "mycopy1" to "mynewfilename."**

```
beaglebone$ mv mycopy1 mynewfilename
```

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

20. **Move "mynewfilename" from the "subdir" directory up one level to the "lab02" directory.**

```
beaglebone$ 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:

```
beaglebone$ mv mynewfilename ..
```

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

21. **Copy everything from the "subdir" directory up one level to the "lab02" directory.**

```
beaglebone$ 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.

22. **Remove "mynewfilename" from the "subdir" directory**

```
beaglebone$ rm mynewfilename
```

23. **Change directories up one level to the "lab01" directory**
24. **Remove the "subdir" directory and all of its contents (recursive removal)**

```
beaglebone$ rm -R subdir
```

25. **Exit from secure shell**

```
beaglebone$ exit
```

(Page intentionally left blank)

Module 02: Application Build and Debug

Introduction

Code Composer Studio version 5 is the current iteration of Texas Instrument's integrated development environment. This IDE can be used to program any of TI's processors from the low-cost, ultra-low-power MSP430 all the way up to the ultra-performance Arm Cortex-A8 devices such as the AM335x.

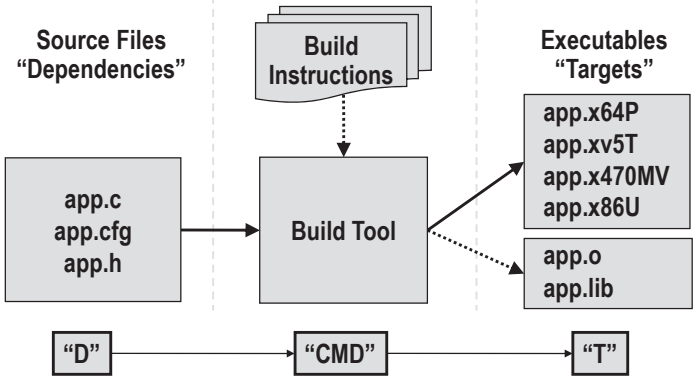
CCSv5 is based on eclipse, an open-source IDE. This provides many advantages. Firstly, customers who are familiar with other eclipse-based IDEs will have less learning curve, and those who are not are likely to run into eclipse-based IDEs in the future. Secondly, there is a large community of eclipse plugins that can be used with CCSv5. Also, eclipse runs on both Windows and Linux host machines.

This module will introduce the basic features of CCSv5 and provide details on setting up a debugging environment.

Module Topics

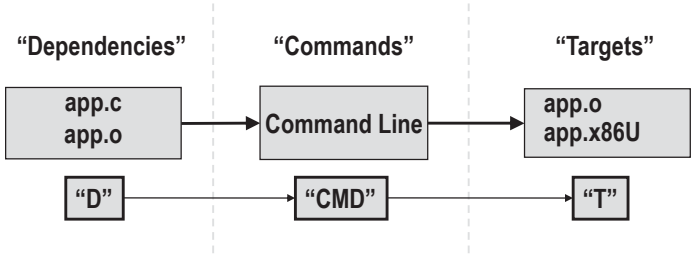
Intro to Makefiles

Build Overview



- Build Tool Goals:
  1. Build ***executable*** (target) from ***input files*** (dependencies) using ***build instructions*** (commands)
  2. Build for ***multiple targets*** at once (e.g. ARM, X86, DSP)
- Solution: command line (e.g. cl6x, gcc) or ***scripting tool*** (gMake, etc.)

Command Line (Examples 1-2)



- Example 1: create an ***object*** file (app.o) from an input file (app.c)

```
gcc –g –c app.c –o app.o
```

- Example 2: create an ***executable*** (app.x86U) from an object file (app.o)

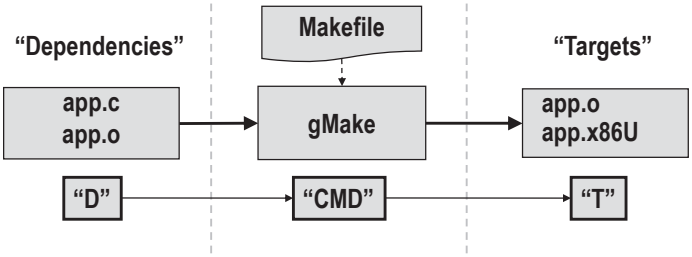
```
gcc   –g   app.o   –o   app.x86U
```

command *flag* *dependency* *flag* *target*

-c = compile only
-g = build with debug enabled
-o = output filename

- Might be ***more convenient*** to place commands in a ***script/batch file…makefile…***

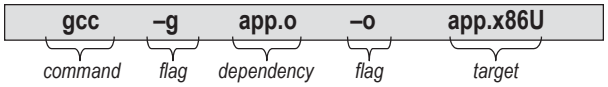
Basic Makefile with Rules



- One of the more common *"scripting"* tools is GNU Make, aka gMake, aka Make…
- gMake uses "rules" to specify build commands, dependencies and targets
- Generically, a ***RULE*** looks like this:

```
TARGET : DEPENDENCY
[TAB]    COMMANDS…
```

- Remember Example 2? Let's make this into a simple Makefile rule:

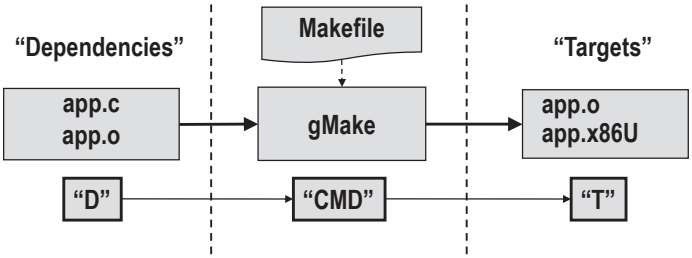


- Becomes….

```
app.x86U  :  app.o
          gcc  -g  app.o  -o  app.x86U
```

RULE

Creating Your First Makefile



Command Lines

```
gcc –c –g app.c –o app.o
gcc –g app.o –o app.x86U
```

Makefile

```
# Makefile for app.x86U (goal)

app.x86U : app.o
      gcc –g app.o –o app.x86U

app.o : app.c
      gcc –g –c app.c –o app.o
```

User-Defined Variables & Include Files

- ***User-defined variables*** **simplify your makefile and make it more readable.**
- ***Include files*** **can contain, for example,** ***path statements*** **for build tools. We use this method to place absolute paths into one file.**
- **If "-include path.mak" is used, the "-" tells gMake to keep going if errors exist.**
- **Examples:**

makefile

```
include path.mak
CC := $(CC_DIR)gcc
CFLAGS := -g
LINK_FLAGS := -o

app.x86U : app.o
    $(CC) $(CFLAGS) $^ $(LINK_FLAGS) $@
```

path.mak

```
CC_DIR := /usr/bin/
...
# other paths go here…
```

Using Built-in Variables

- ***Simplify your makefile*** **by using these built-in gMake variables:**
  - **$@ = Target**
  - **$^ = All Dependencies**
  - **$< = 1st Dependency Only**
- ***Scope*** **of variables used is the current rule only.**
- **Example:**

Original makefile…

```
app.x86U: app.o
     gcc –g app.o –o app.x86U
```

Becomes…

```
app.x86U: app.o
     gcc  –g  $^  –o  $@
```

Templated Rules Using "%"

- Using pattern matching (or pattern substitution) can help *simplify your makefile* and help you remove explicit arguments. For example:

Original Makefile

```
app.x86U : app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

app.o : app.c
    $(CC) $(CFLAGS) -c $^ -o $@

main.o : main.c
    $(CC) $(CFLAGS) -c $^ -o $@
```

Makefile Using Pattern Matching

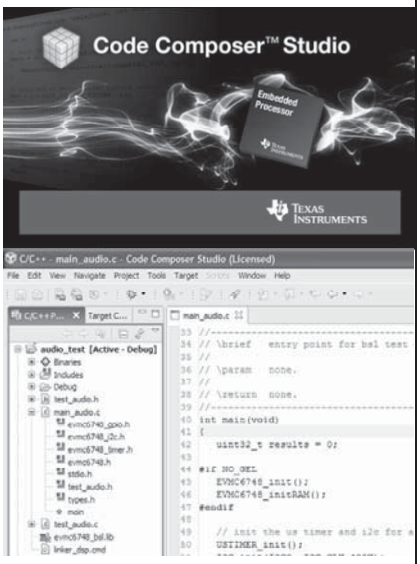
```
app.x86U : app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

%.o : %.c
    $(CC) $(CFLAGS) -c $^ -o $@
```

- The .x rule depends on the .o files being built – that's what kicks off the .o rules
- % is a shortcut for $(patsubst …), e.g. $(patsubst .c, .o)

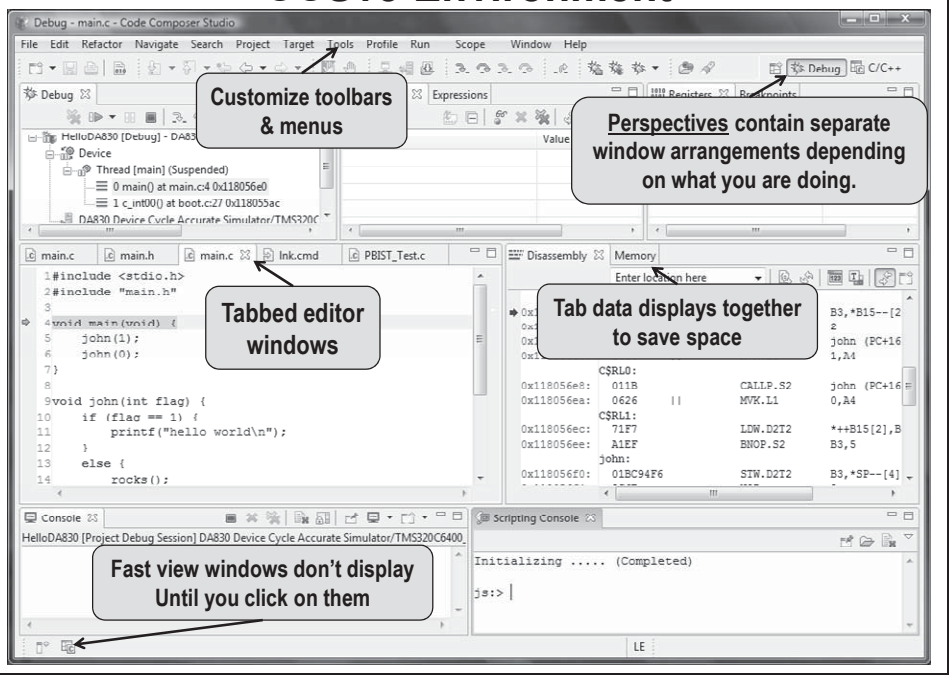
Code Composer Studio Overview

What is Code Composer Studio?

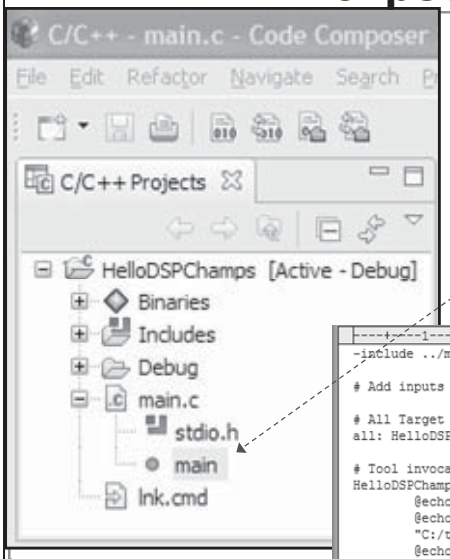
- Integrated development environment for TI embedded processors
  - Includes debugger, compiler, editor, simulator, OS…
- CCSv5 is based on "off the shelf" Eclipse
  - TI contributes changes directly to the open source community
  - Drop in Eclipse plug-ins from other vendors or take TI tools and drop them into an existing Eclipse environment
  - CCSv5 runs on a Windows or Linux host
- Integrate additional tools
  - OS application development tools (Linux, Android…)
  - Code analysis, source control…
- Low cost!



CCSv5 Environment



Eclipse Projects



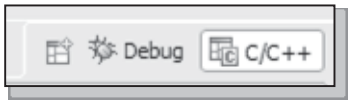
- CCSv5 is PROJECT-centric
- Eclipse uses ***managed makefiles*** as their build scripts – as opposed to ***pjt*** files
- Eclipse projects are folder based
  - "Adding file" copies it to folder
  - "Linking file" references original file
  - Project explorer shows folder contents
- Project explorer lists functions

make file

Perspectives

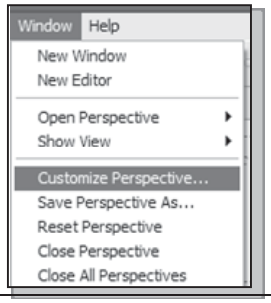
- ***Perspectives*** – a set of windows, views and menus that correspond to a specific set of tasks
- Two ***default perspectives*** are provided with CCSv5:

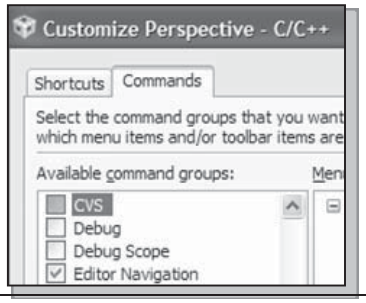
Debug
- Debug Views
- Watch/Memory
- Graphs, etc.



C/C++
- Code Dev't Views
- Project Contents
- Editor

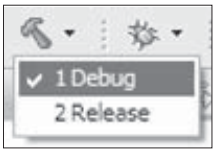
- Users can ***customize perspectives*** and save them:





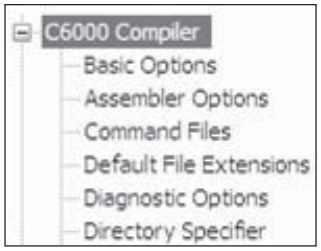
Two Default Build Configurations

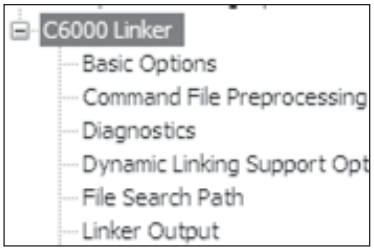
- ***Build Configuration*** **– a set of build options for the compiler and linker (e.g. optimization levels, include DIRs, debug symbols, etc.)**
- **CCSv5 comes standard with *two DEFAULT build configs*: Debug & Release:**



User can create their own config if desired

- **User can modify compiler/linker options via "Build Options":**





Many more details on compiler options in a later chapter…

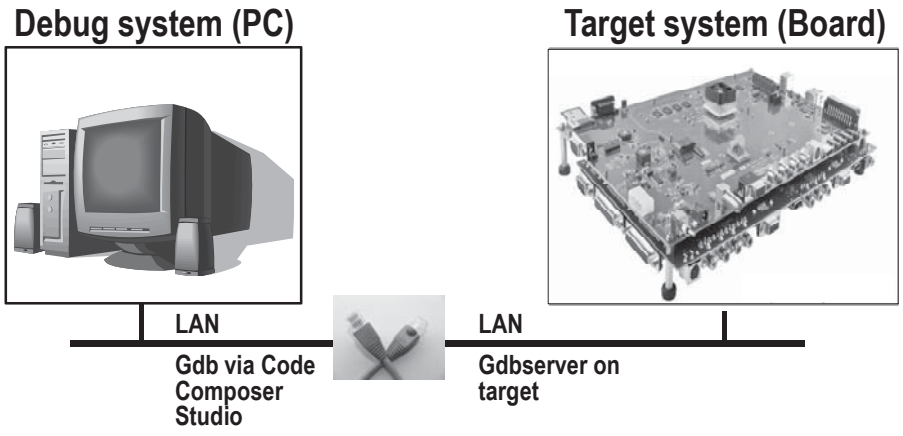
18

Debugging & Debugger's

- **User Mode Debugging**
  - When debugging user mode programs, you often only want to debug – hence stop – one thread or program.
  - GDB (GNU Debugger) works well for this. (GDB discussed on next slide)
  - Connection is usually Ethernet or serial-port
- **Kernel Mode Debug**
  - Debugging kernel code requires complete system access.
  - You need KGDB (Ethernet) or scan-based (JTAG) debuggers for this.
- **A debugger lets you**
  - Pause a program
  - Examine and change variables
  - Step through code
- **Code Composer Studio (CCSv5)**
  - Latest version of TI's graphical software debugger (i.e. IDE)
  - IDE – integrated development environment: which combines editing, building and debugging into a single tool
  - Built on Eclipse platform; can install on top of standard Eclipse
  - Allows debugging via GDB (Ethernet/serial) or JTAG (scan-based)
  - Free license for GDB debugging

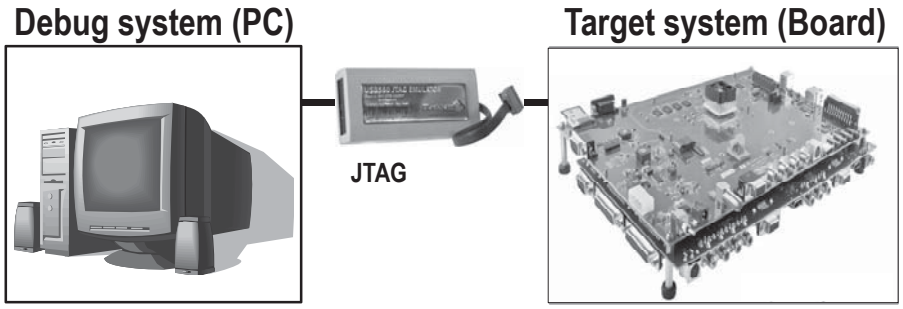
Linux application debug setup

- **Physically the connections of a typical linux application debug system are shown below.**



Linux kernel and driver debug setup

- **Physically the connections of a typical uboot, linux kernel or driver debug system are shown below.**



GDB : GNU Debugger

- **Open source debugger that is often supplied with the toolchain**
- **In this workshop, it's included in the Code Sourcery package**
- **GDB has a client/server nature:**

GDB Server:

- First start *gdbserver*, specifying connection and app to be debugged
- Server then runs your app, following gdb commands

Tera Term

EVM> **gdbserver 192.168.1.122:10000 myApp**

Listening on port 10000
Remote debugging from host 192.168.1.1

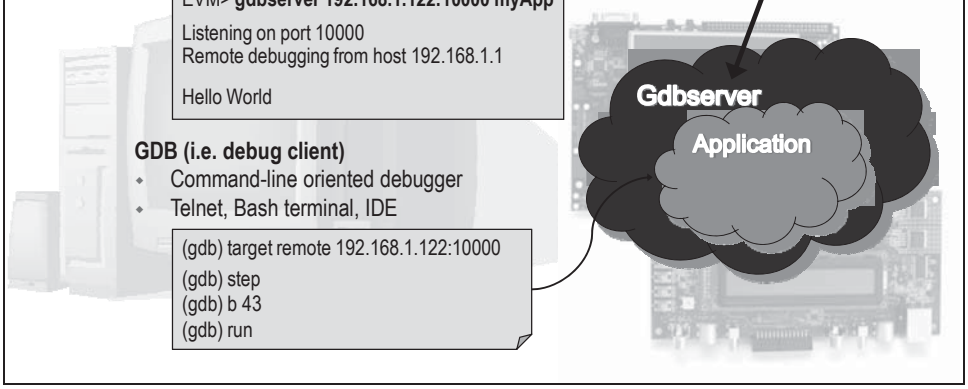
Hello World

GDB (i.e. debug client)

- Command-line oriented debugger
- Telnet, Bash terminal, IDE

```
(gdb) target remote 192.168.1.122:10000
(gdb) step
(gdb) b 43
(gdb) run
```

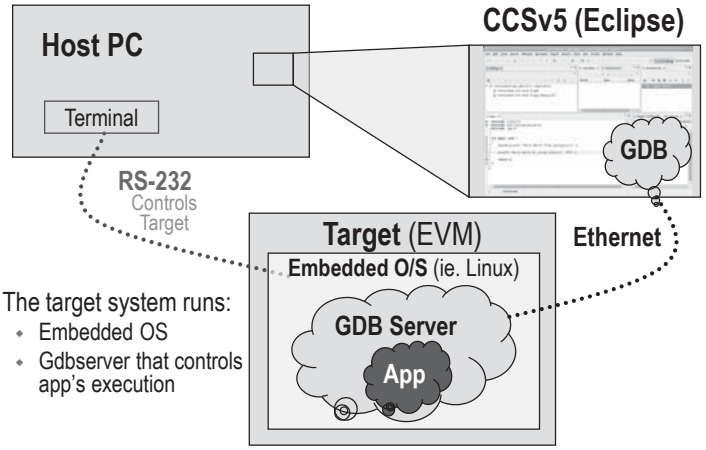
GDB Server
Runs your app for you, based on the GDB commands you send



Graphical Debug with GDB

Eclipse (CCSv5) and other IDE's can translate actions to GDB cmds

- Other than starting gdbserver, it means we don't have to know GDB syntax
- The debug (host) system runs:
  - Its own OS (Linux, Windows, etc.)
  - Debugger IDE (optional) and Gdb
  - Terminal app (Tera Term, putty, etc.) to control the target/evm environment
- The target system runs:
  - Embedded OS
  - Gdbserver that controls app's execution



CCSv5 Licensing & Pricing

- **Licensing**
  - **Wide variety of options (node locked, floating, time based…)**
  - **All versions (full, DSK, free tools) use same image**
  - **Updates readily available via the internet**
- **Pricing**
  - **Reasonable pricing – includes FREE options noted below**
  - **Annual subscription – $99** *($149 for floating)*

Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 120 day limit (all EMU)	FREE	
Platinum Bundle	EVM, sim, XDS100 use	FREE ☺	
Platinum Node Lock	Full tools tied to a machine	$495 (1)	$99
Platinum Floating	Full tools shared across machines	$795 (1)	$159
Microcontroller Core	MSP/C2000 code size limited	FREE	
Microcontroller Node Lock	MSP/C2000	$445	$99

☺ - recommended option: purchase Dev Kit, use XDS100v1-2, & Free CCSv5

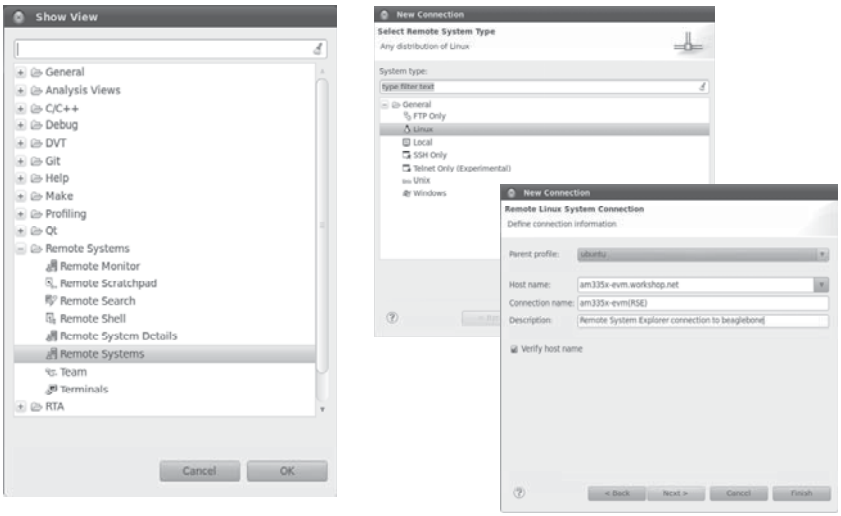
Texas Instruments

20

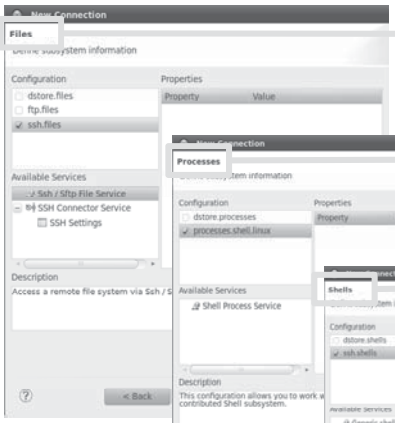
Setting up a CCS5 GDB Session

Remote Systems View (1)

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



Remote Systems View (2)

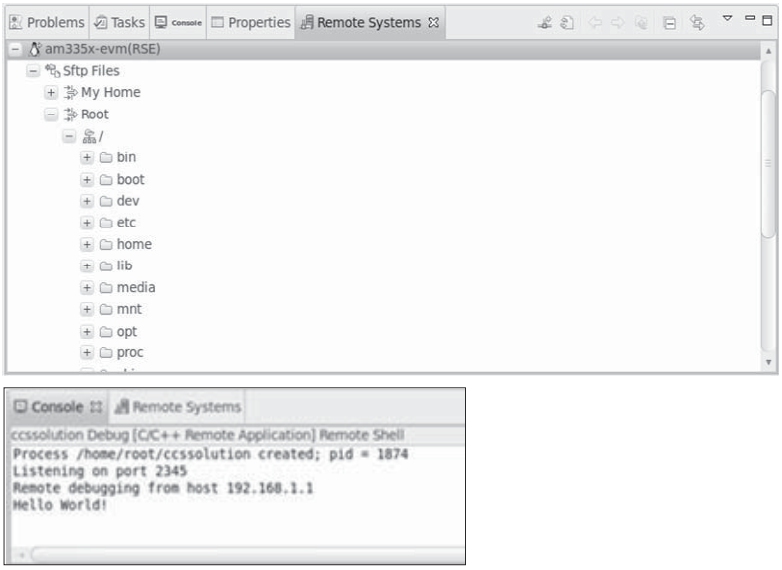


Files: tells CCS how to view remote files on and transfer files to remote system

Processes: tells CCS how to view and manipulate remote processes (applications)

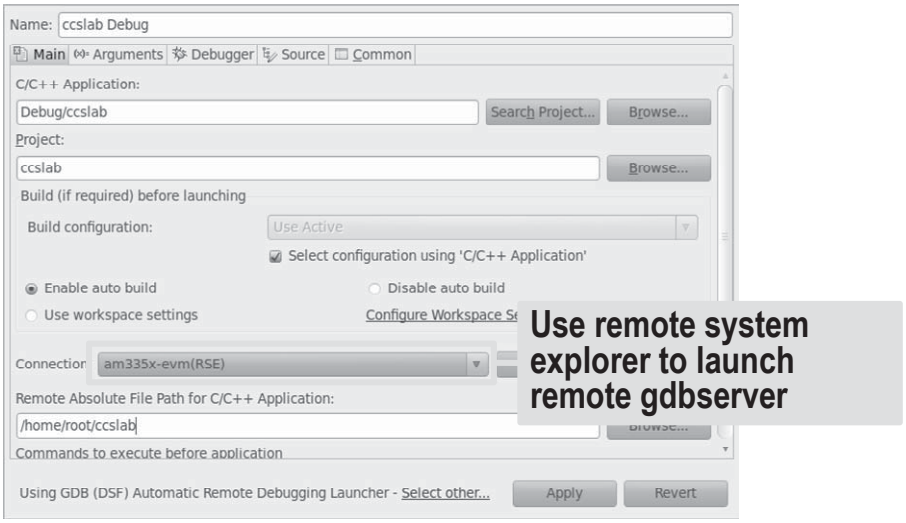
Shells: tells CCS how to launch remote applications for debugging

Remote System View – Files / Terminal



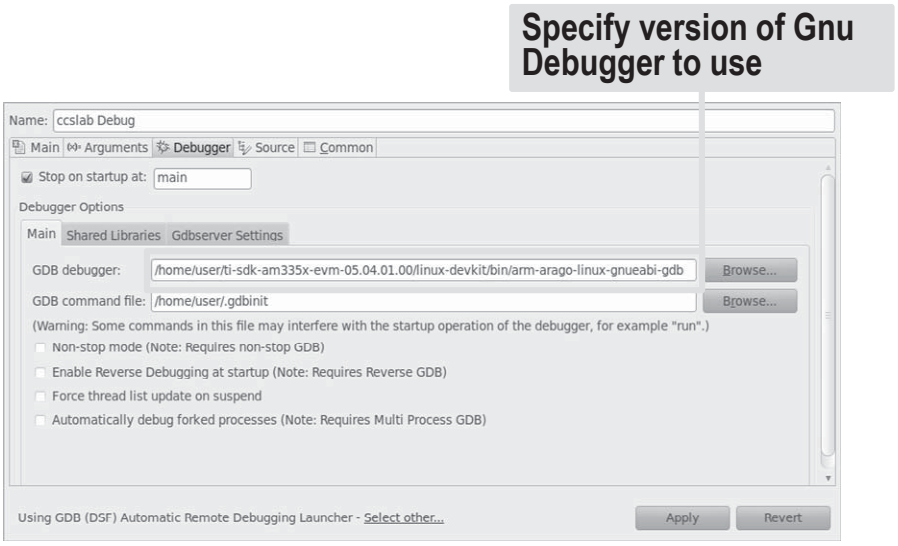
Setting Up CCSv5 Application Debug

Run → Debug Configurations → New Launch Configuration

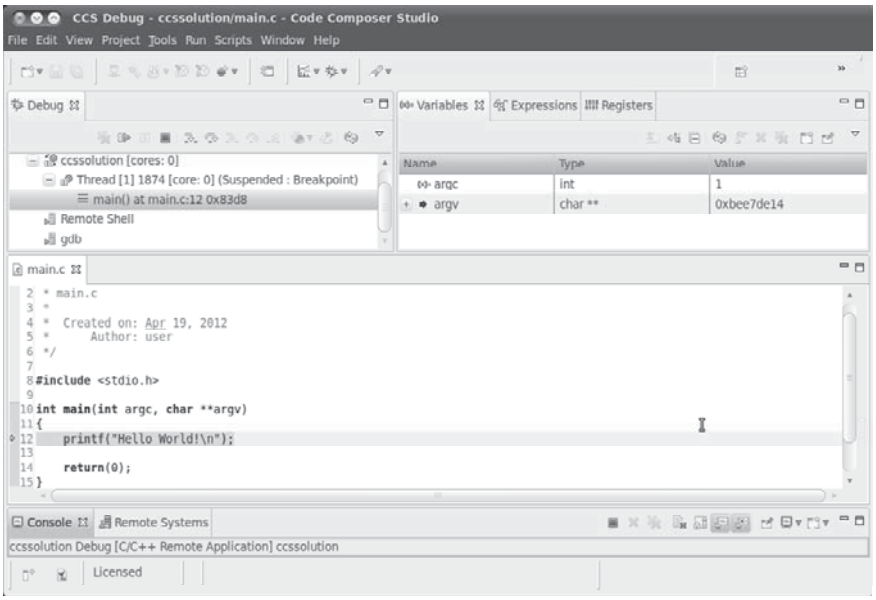


Use remote system explorer to launch remote gdbserver

Setting Up CCSv5 Application Debug



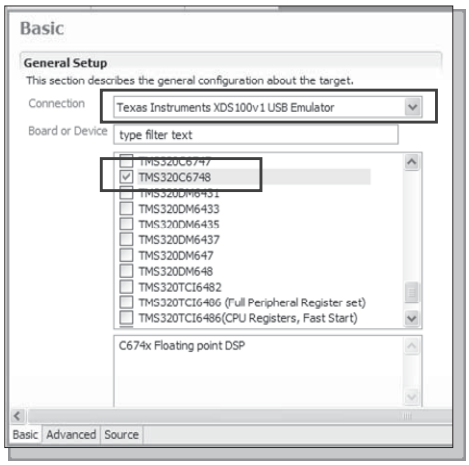
CCSv5 Attached Remote Debugger



Setting up a CCS5 JTAG Session

Creating a New Target Config File (.ccxml)

- ***Target Configuration*** **– defines your "target" – i.e. emulator/device used, GEL scripts (replaces the old CCS Setup)**
- **Create user-defined configurations (select based on chosen board)**



16

Lab 2: Code Composer Studio Debug

This last lab exercise explores using CCSv5 (i.e. Eclipse) for building and debugging our Linux applications. First, we'll install CCSv5; then set our project and remote debugging; then finally run/debug our 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 (CCSv5/Eclipse) and the target (Linux application running on the ARM processor). We'll find the gdb executable along with our build tools in the provided Texas Instruments Software Development Kit (SDK) for Sitara processors.

Here are a couple good references that you may want to refer to in the future:

CCSv5 wiki: http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5

Linux Debug in CCSv5: http://processors.wiki.ti.com/index.php/Linux_Debug_in_CCSv5

CCSv5 Installation

We've installed CCSv5 into the `/home/user/ti` folder

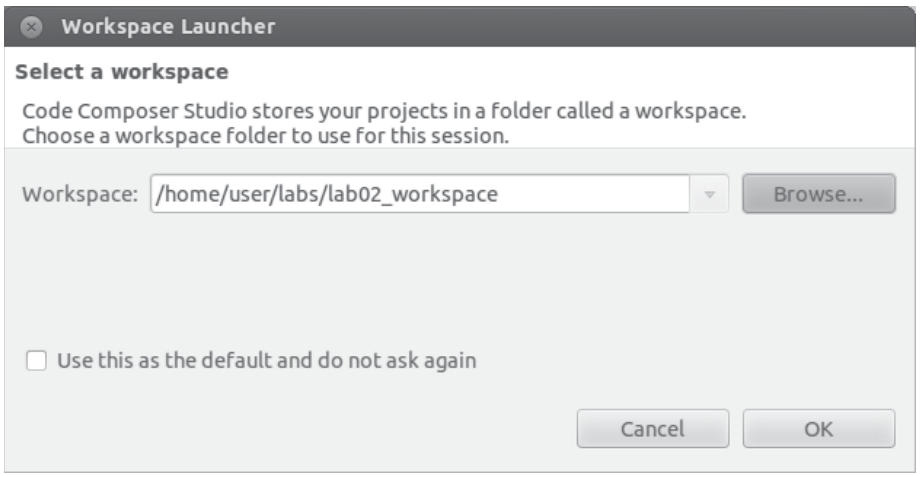
Module Topics

Create Project

1. **Start CCSv5 from the Ubuntu desktop.**

2. **Select the "/home/user/labs/lab02_workspace" Workspace**

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

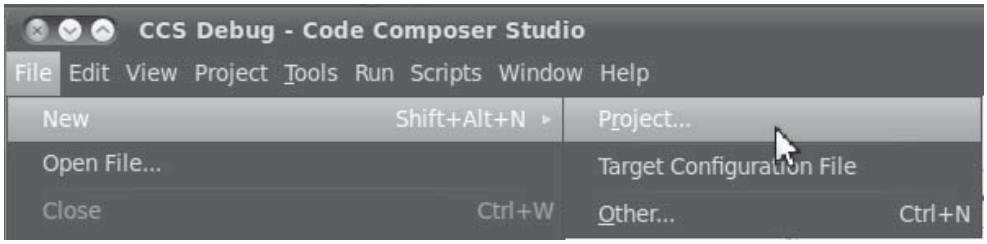


   The workspace contains general information that is outside the scope of individual projects such as the debugging connections that have been configured and the general settings for the IDE. Each lab exercise is organized into its own workspace for this workshop.

   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

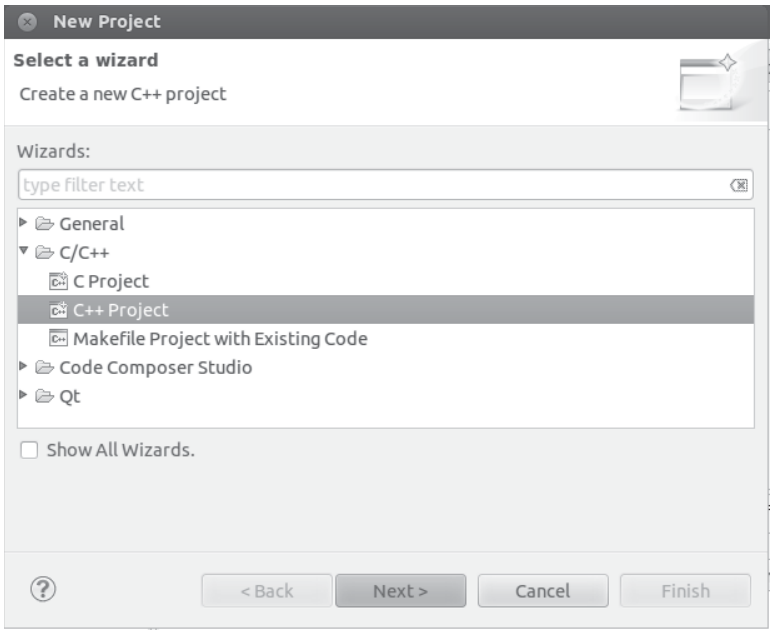
3. **Create a new project in the `/home/user/labs/lab02_workspace` directory**



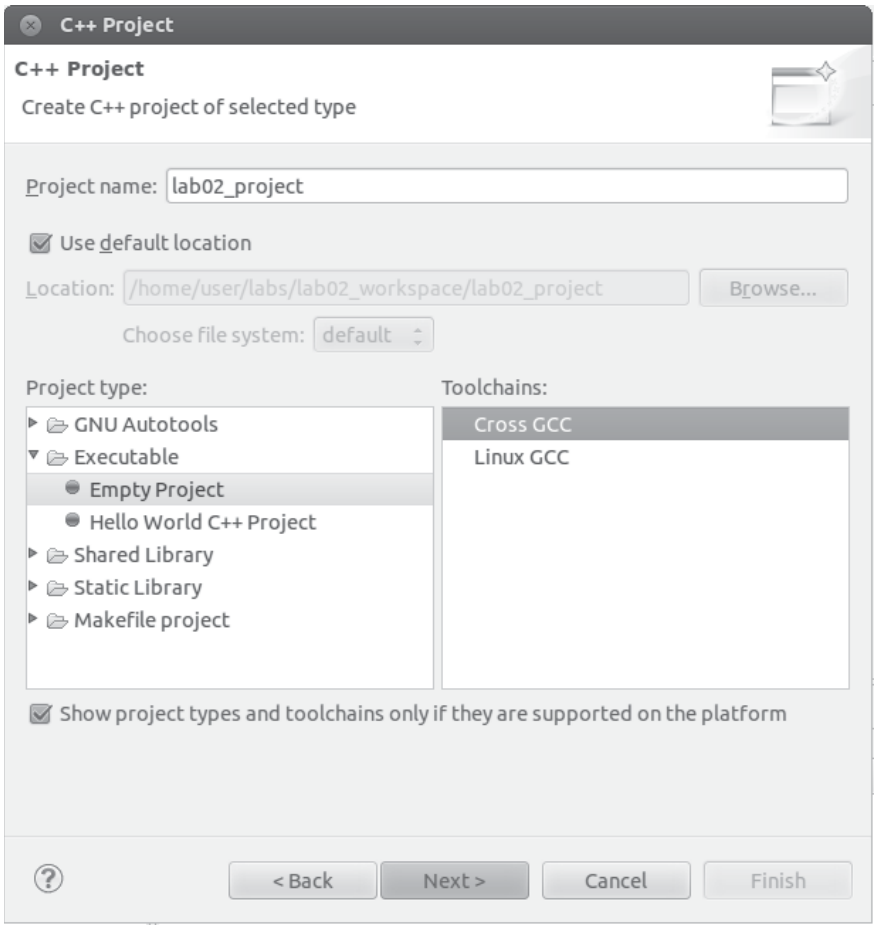
Eclipse provides many different types of projects. The project type selected will affect which build tools are used to build your project

- **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. A CCS Project sets up JTAG-based debugging.
- **C/C++→C++ Project** (or C Project) – Also uses Eclipse's managed make capability, but sets up GDB-based debugging.
- **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++ → C++ Project" as shown in the following diagram. Select in the tree and then press "Next."

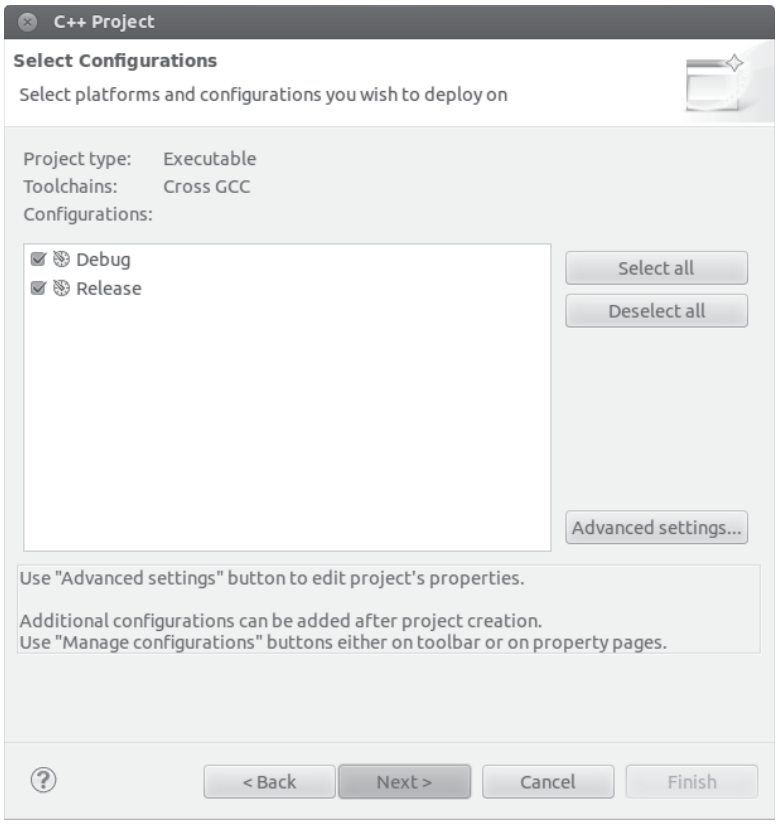


On the next screen, name the project “lab02_project” and select “Empty Project” from the “Executable” project type. Use the default Toolchain: “Cross GCC”



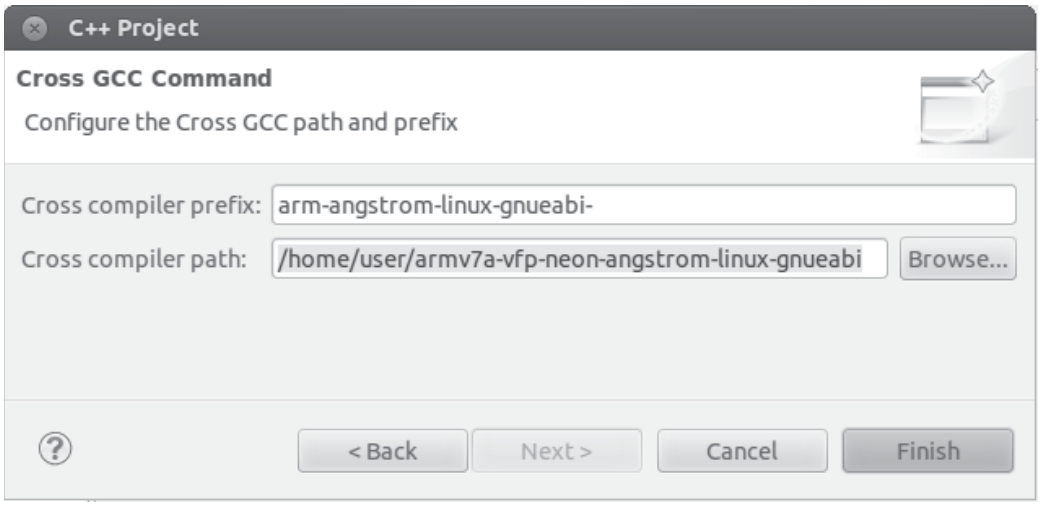
NOTE: CCS will default to the “GNU Autotools” Project Type in the left pane, which also has an “Empty Project” type, but not the “Cross GCC” Toolchain. If you do not see “Cross GCC” Toolchain, be sure to select “Executable” Project type in the left pane.

Press “Next>”



Press “Next>”

You will need to provide the location and name of the cross-compile toolchain on the system. It is recommended that you use the “Browse..” button to select the “Tool command path” as a single typo will cause problems later. The “Tool command prefix” is everything in the command that comes before “gcc” for the cross-gnu compiler chain tools.



Note that this path is actually a symbolic link to:

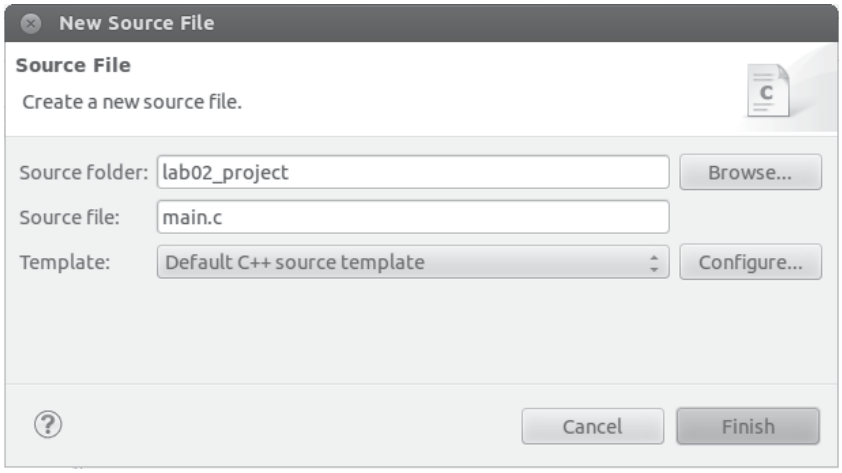
/usr/local/oecore-i686/sysroots/i686-andstromsdk-linux/usr/bin/armv7a-vfp-neon-angstrom-linux-gnueabi

The bitbake tool that is used to build the Angstrom distribution, including the cross compiler chain, uses a large number of subdirectories for organization. In order to guarantee that the compiler can find all of the standard libraries it requires (such as "stdio.h") it needs to be installed to the expected location, which is the long set of paths above. Fortunately, a simlink allows us to create a simpler reference to this location.

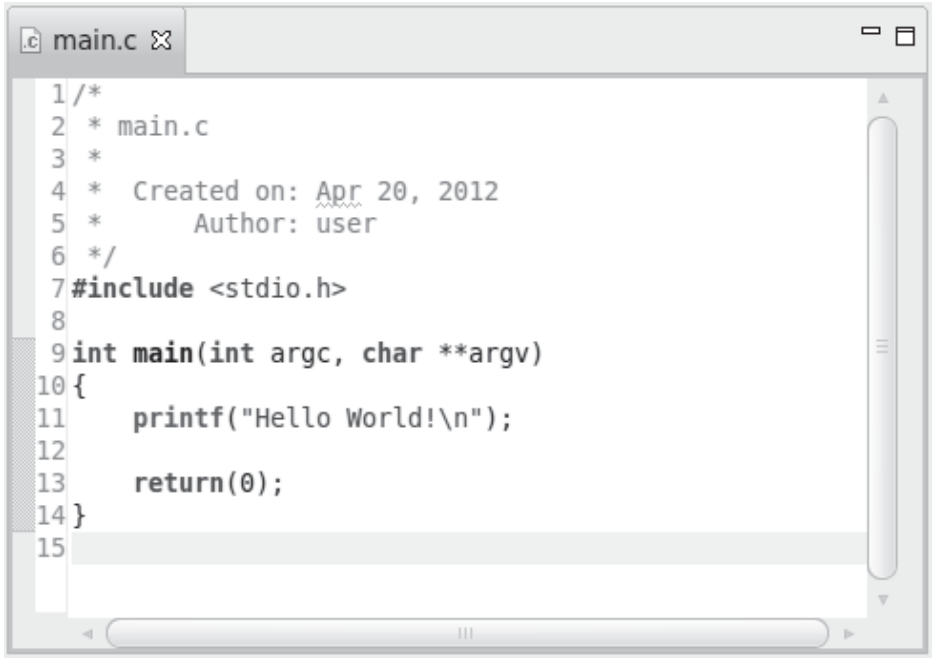
Press "`Finish.`"

4. **Add a file "main.c" to your project**

   File→New→Source File



5. **Type the following short program into main.c and save**



6. **Build your program**

   Project→Build All (ctrl-b)

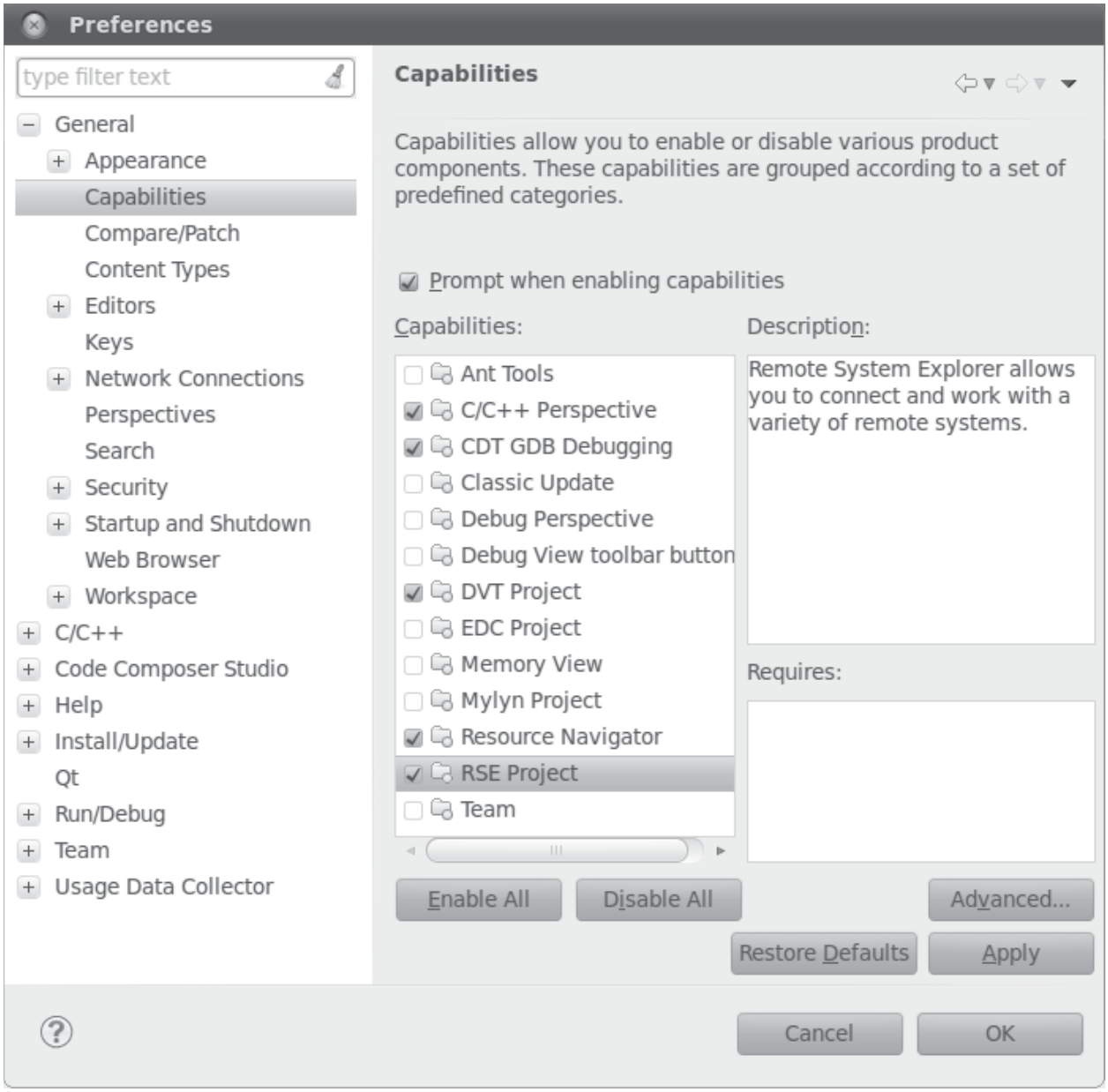
Setup CCSv5 Remote System Explorer Connection

IMPORTANT! By default CCS does not enable “CDT GDB Debugging” or “RSE Project” capabilities, both of which we will need for debugging as outlined in this lab.

7. **Enable the “CDT GDB Debugging” or “RSE Project” capabilities so that we can access GDB debugging from CCSv5.**

   Open the Capabilities tab in the CCSv5 Preferences dialog.

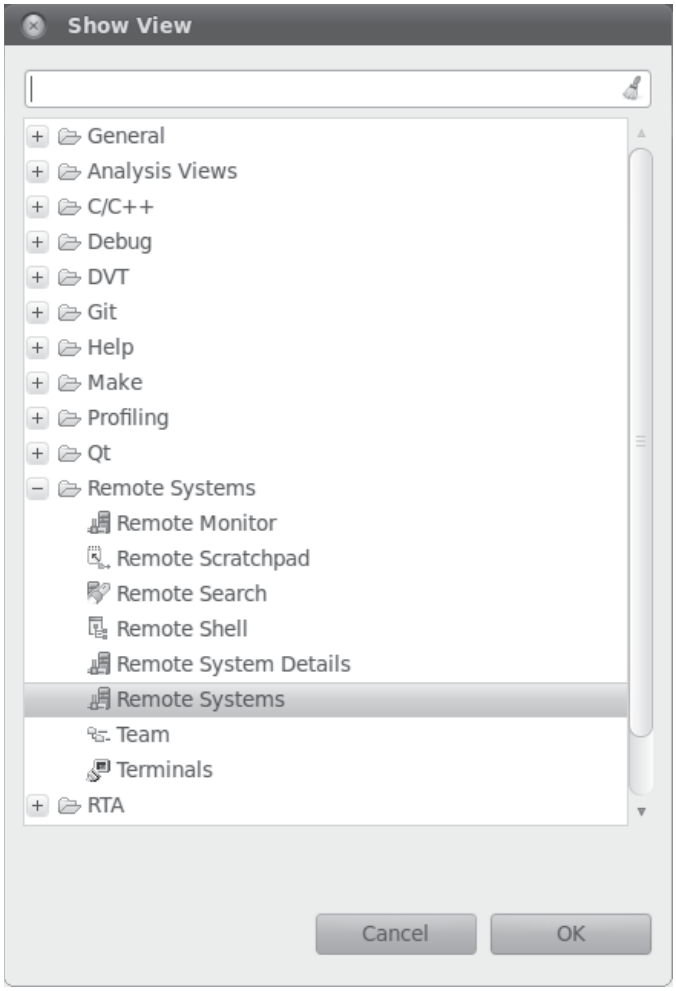
   `Window → Preferences → General → Capabilities`



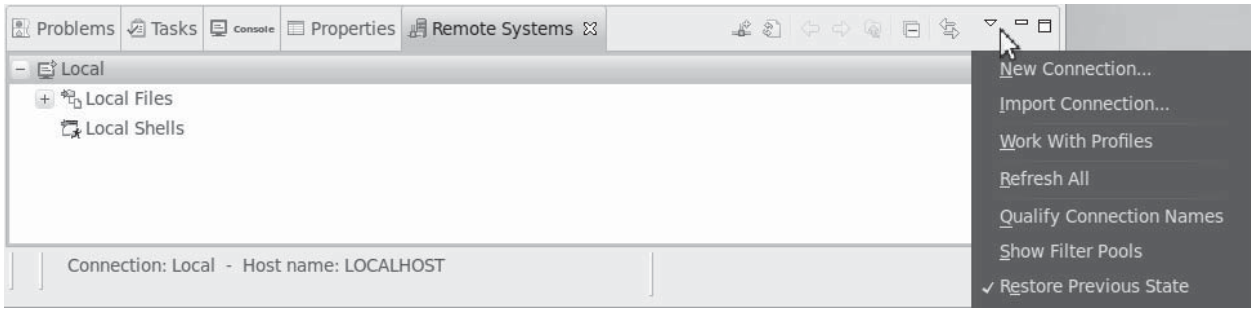
Enable *CDT GDB Debugging and RSE Project*, and then click *OK*.

8. **Open the remote systems view**

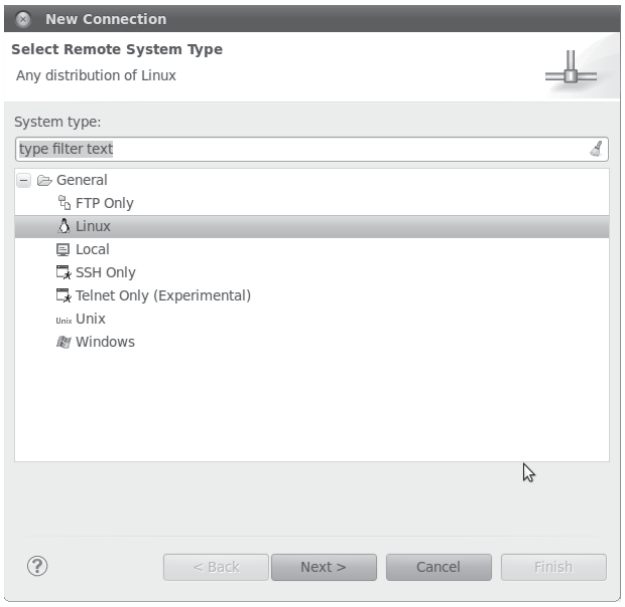
   Window → Show View → Other… → Remote Systems → Remote Systems



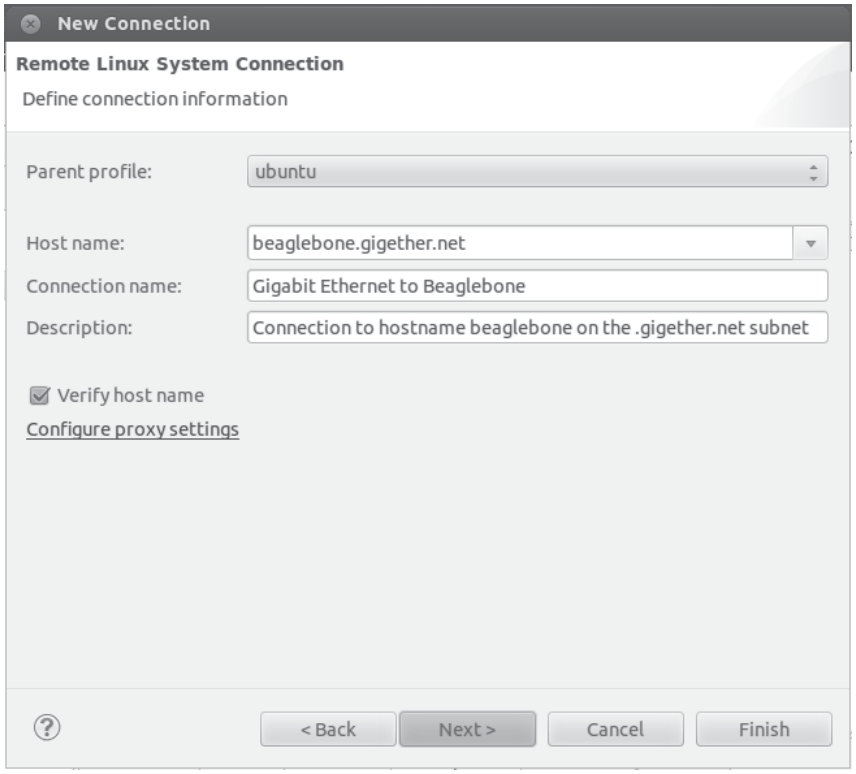
9. **Configure a new connection using the Remote Systems menu or by pressing the new connection button**



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



Remote System Type: `Linux`

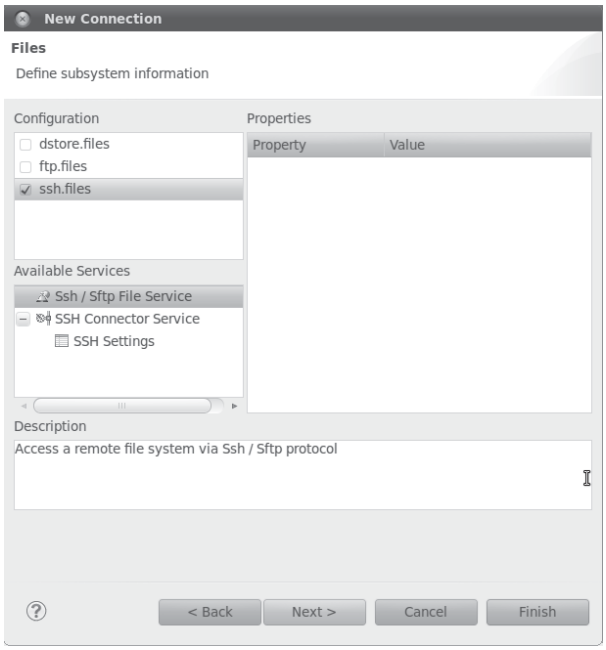


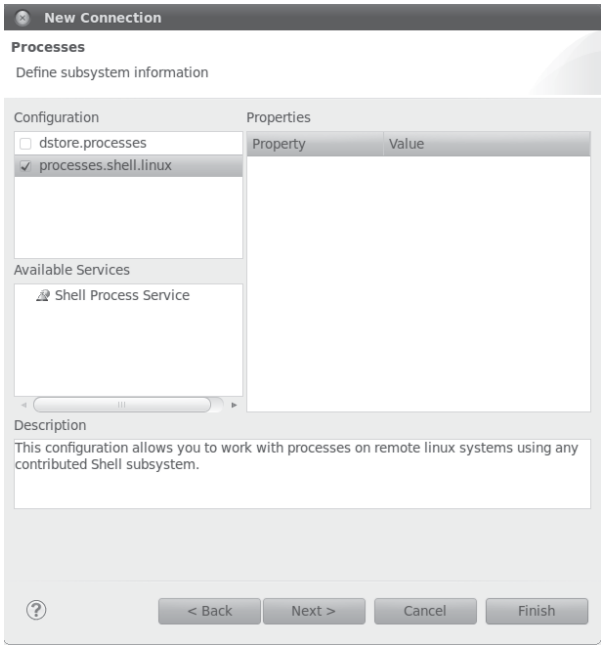
Host Name: `beaglebone.gigether.net`

Connection name: `Gigabit Ethernet to Beaglebone`

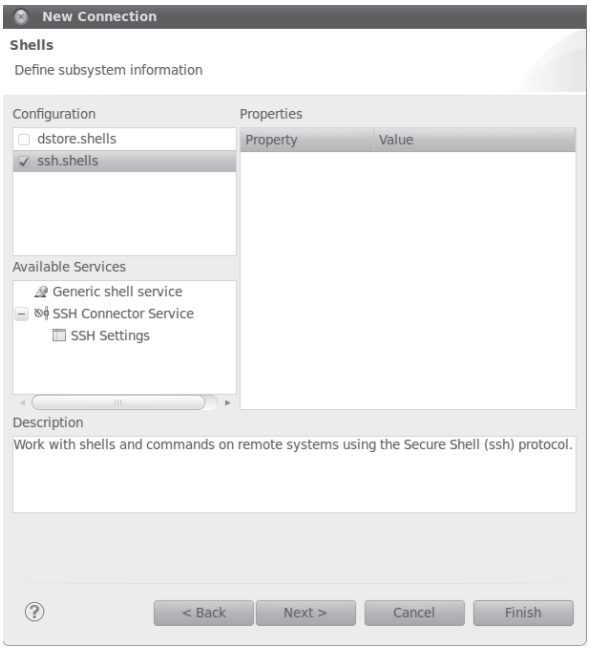
Description: `Connection to hostname beaglebone on the .gigether.net subnet`

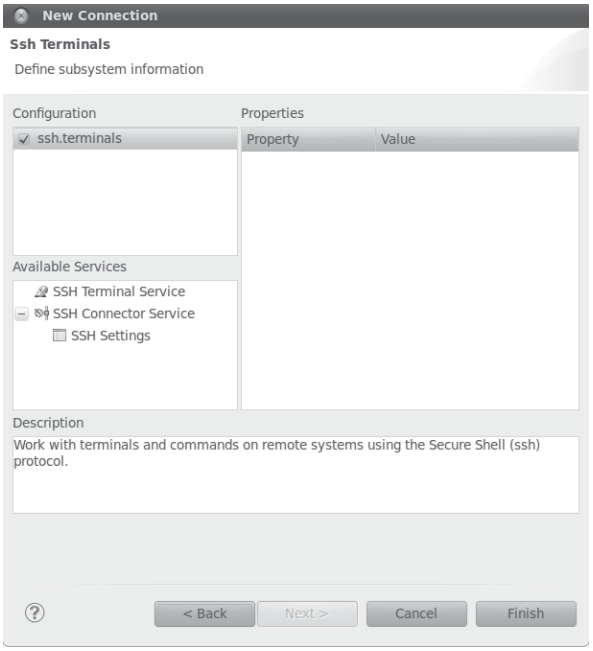
NOTE: `Press "Next>" and not "Finish"`





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

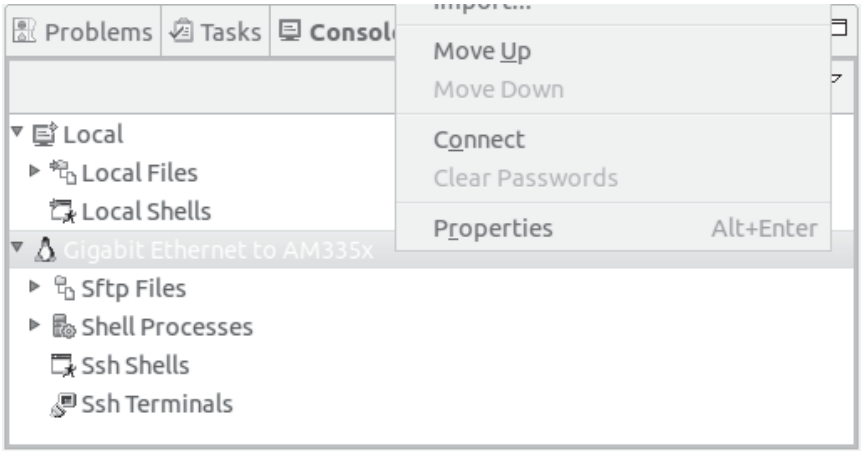




On the fifth screen (shells) choose “ssh.shells” and on the sixth screen (ssh Terminals) choose “ssh.terminals.” You may press “Finish” on the sixth screen.

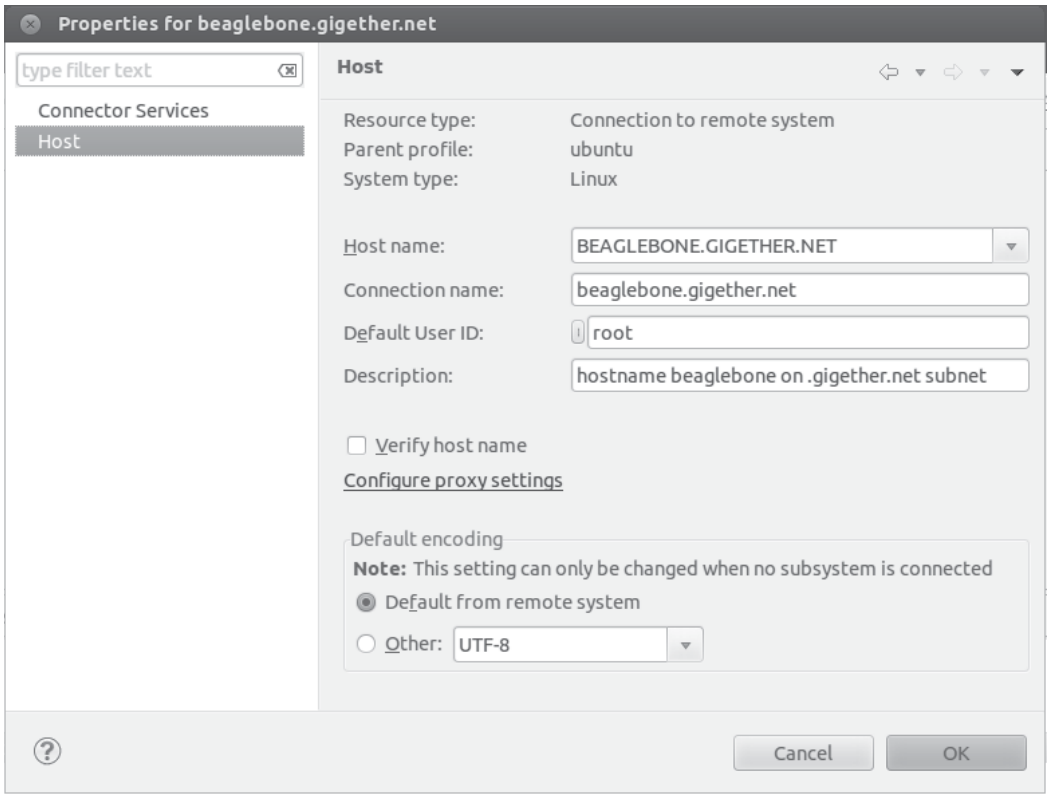
11. Change the ssh login user ID to “root”

Right-click “Gigabit Ethernet to Beaglebone” 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.

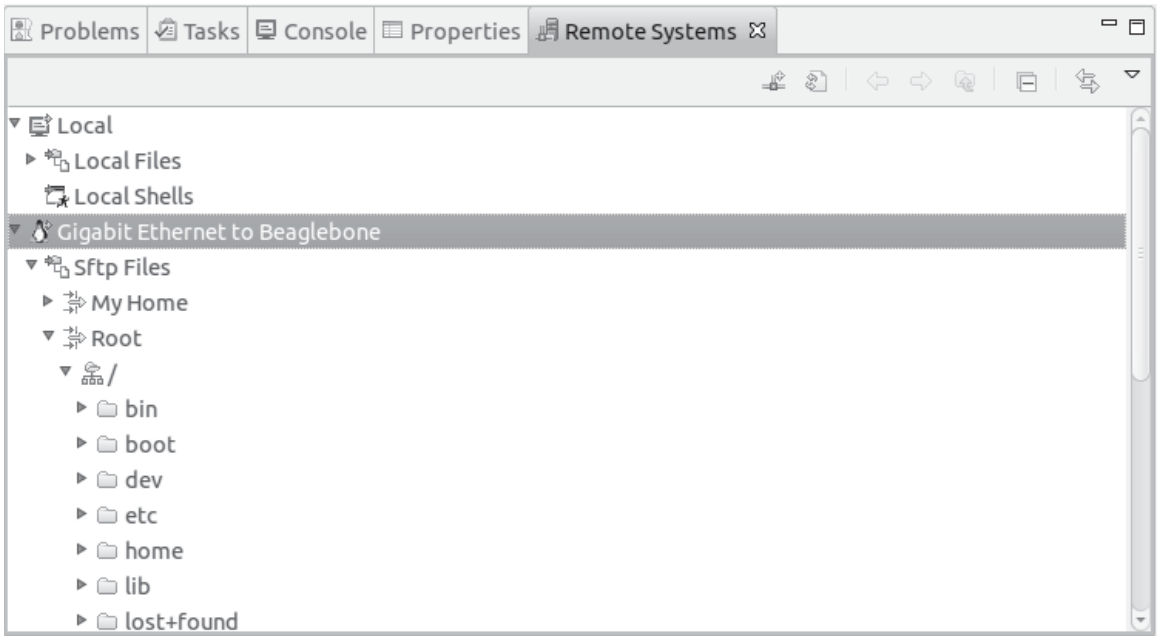


Press OK

12. **Test connection**

    Expand the “Gigabit Ethernet to Beaglebone” connection as shown below.

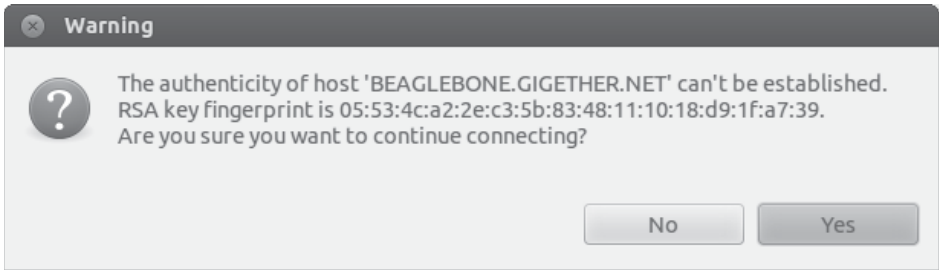
    When you expand the “Root” subtree, you may be prompted as shown in the following steps.



13. **If needed – ssh authentication**

    You may get a message indicating the ssh service does not recognize the RSA key fingerprint of the Beaglebone Black. This is the same message that you should have seen in step **Error! Reference source not found.** from the terminal, and it will only appear if you did not generate a new key in that step.

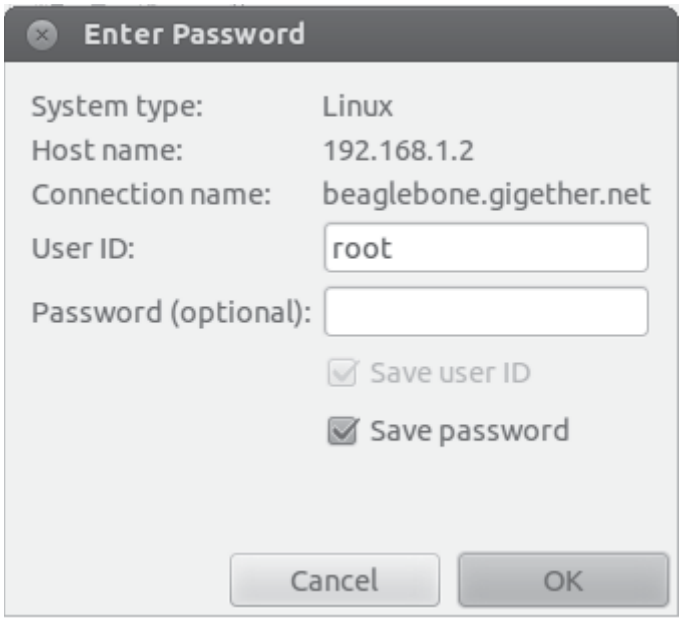
    If it appears, press “Yes” to continue connecting and you should not see it a second time.



14. If needed – blank password

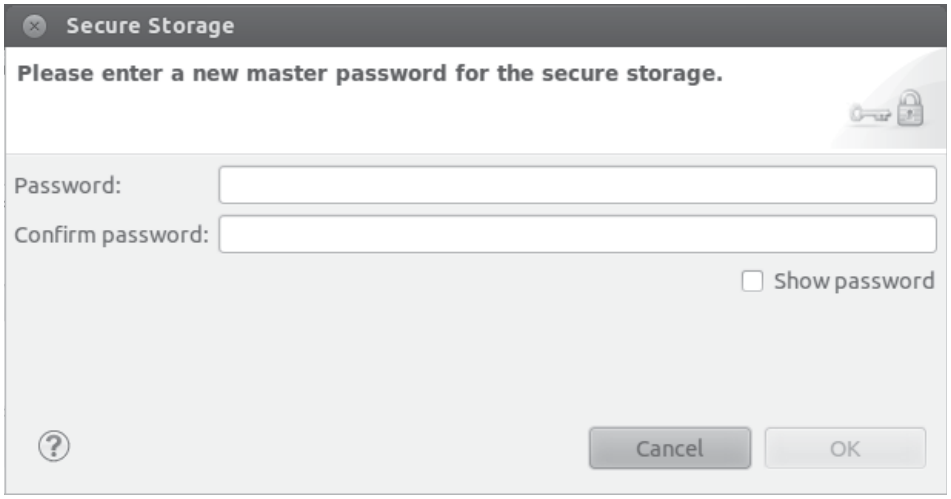
If prompted for a password, leave blank, select “Save Password” and press “OK.”

In some cases this window does not allow you to enter a blank password. If that is the case, proceed and when prompted for a password during the debug connection, it should allow a blank password.



15. If needed – blank secure storage password

You may get a message box prompting you for a secure storage password. If so, just press “Cancel.”



16. **(Optional) Configure Ethernet-over-USB connection**

    You can do this by repeating steps 9-15 using the following

    Remote System Type: `Linux`

    Host Name: `beaglebone.etherusb.net`

    Connection name: `Ethernet-over-USB to Beaglebone`

    Description: `Connection to hostname beaglebone on the .etherusb.net subnet`

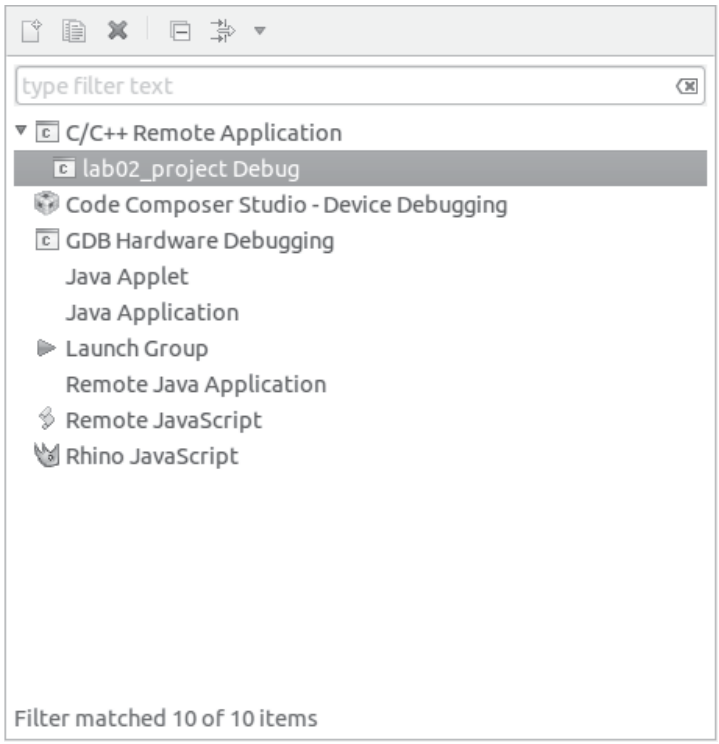
Setup CCSv5 Debug Configuration

17. **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.” You may name the configuration anything you find descriptive. Below it has been named “lab02_project Debug” (the default)



18. **Setup the configuration:**

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

    Name: **lab02_project Debug**

    Project: **lab02_project**

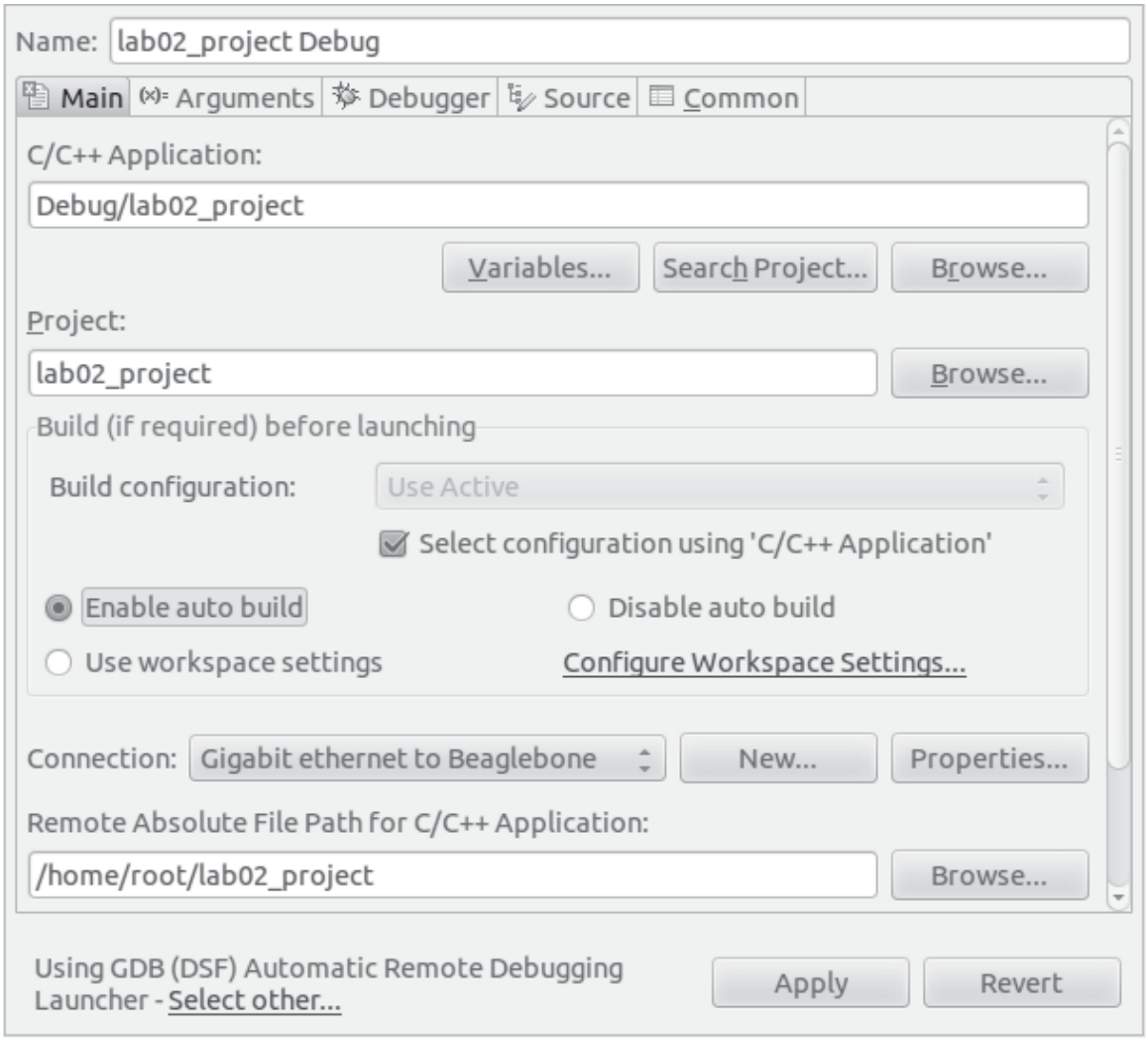
    C/C++ App: **Debug/lab02_project**

    Enable **the auto-build option.**

    Connection: **Gigabit Ethernet to Beaglebone**

    Remote Path: **/home/root/lab02_project**

    … but don't close the dialog, we're not done yet …

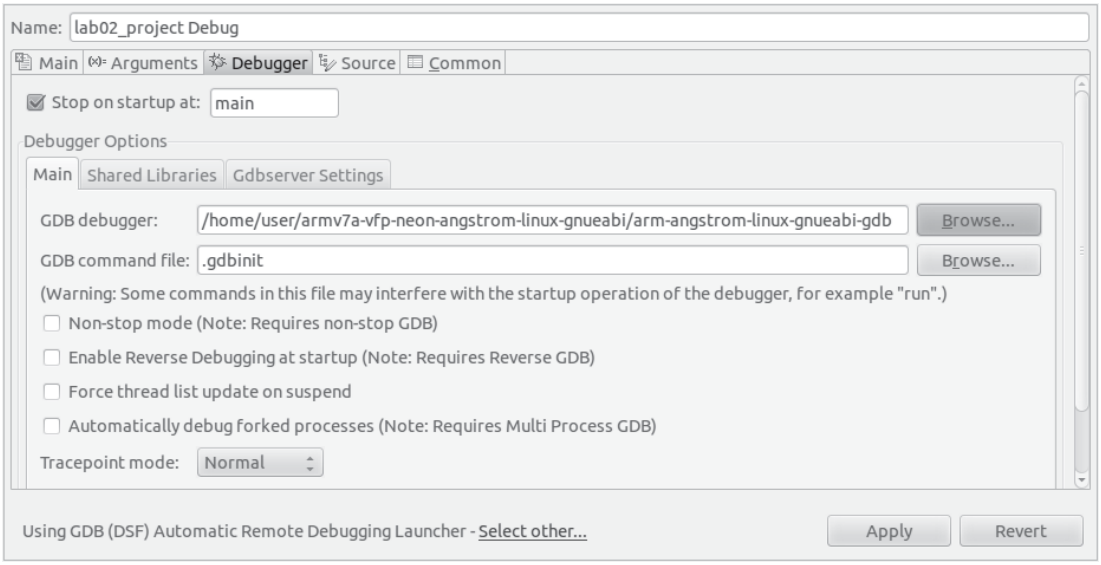


19. **On the *Debugger Main* tab, specify the GDB debugger.**

    We are using the GDB debugger from the TI SDK, so browse for the correct gdb client executable.

    ```
    /home/user/labs/ armv7a-vfp-neon-angstrom-linux-gnueabi/
    arm-angstrum-linux-gnueabi-gdb
    ```

    You may wish to view "`/home/user/.gdbinit`" which had to be added for gdb/CCS to correctly locate shared object libraries. Note that if you browse to this file, it is hidden. You will need to right-click in the file browser and select "show hidden files."



    The gdb initialization file at "/home/user/.gdbinit" contains a single line:

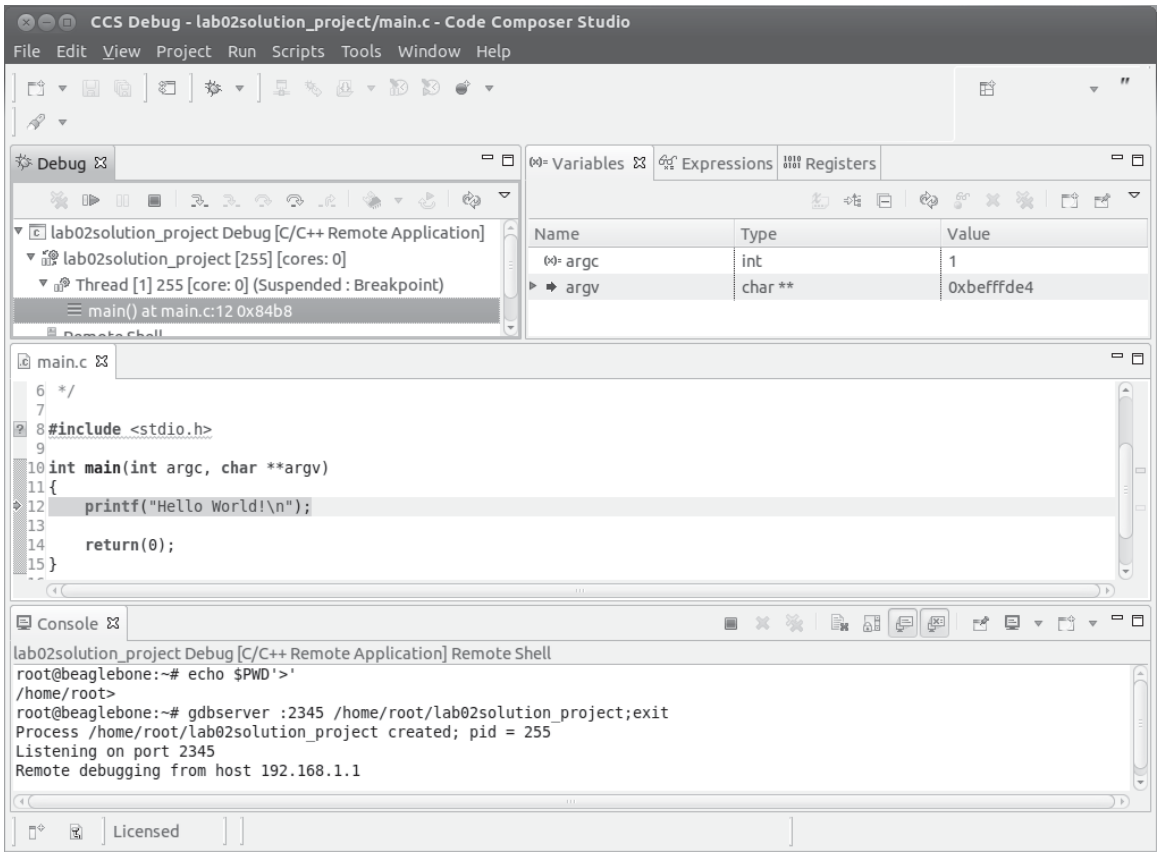
    ```
    set sysroot /home/user/labs/lab00_targetlibs
    ```

    This instruction tells gdb to search for Linux shared libraries in the `/home/user/labs/lab00_targetlibs` directory. This directory was created by copying the directories "/lib" and "/usr/lib" from the target filesystem.

    If this step were skipped, the debugger would still work, but a number of warnings would be generated indicating that the debugger cannot locate various shared libraries.

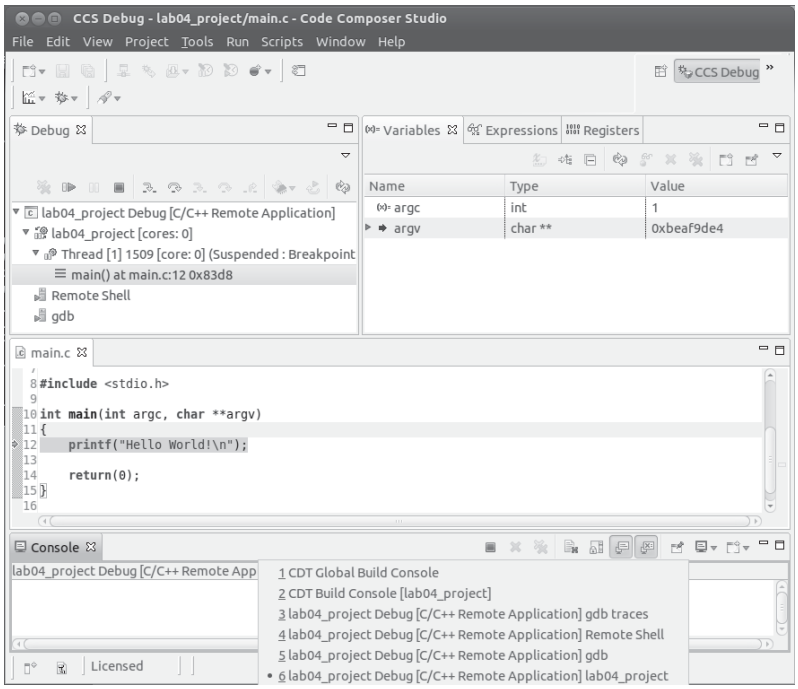
20. **Press the Debug Button to begin Debugging your application.**

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



22. 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.

23. Run through the application once

You may press the start button  or use Run→Resume (F8)

24. 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 launch a new debugging session by using the pull-down and selecting "lab02_project Debug." (The previous debugging session ends when the application exits.)

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

25. 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)

26. 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

Module 03: Linux Scheduler

Introduction

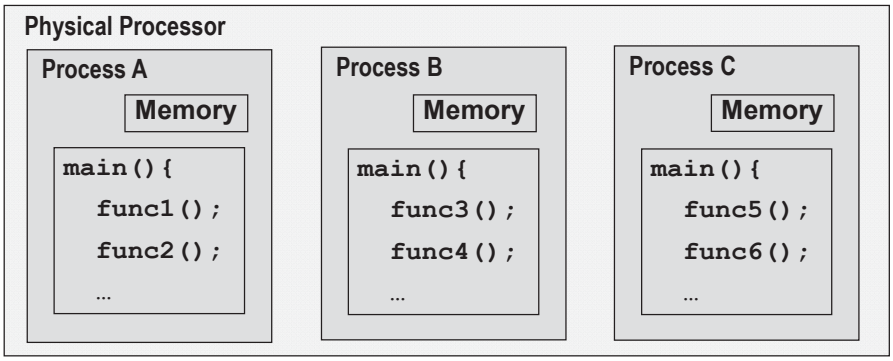
In this lab exercise you will explore the Linux Scheduler using POSIX threads (pthreads) and semaphores. You will create threads that print a message indicating which thread is running. In the first exercise, you will use standard time-slicing threads without semaphores. In the next exercise you will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

Linux Processes

What is a Process?



Scheduling Methodologies

Time-Slicing Threads

Scheduler shares processor run time between all threads with greater time for higher priority

- ✓ No threads completely starve
- ✓ Corrects for non-"good citizen" threads
- × Can't guarantee processor cycles even to highest priority threads.
- × More context switching overhead

Realtime Threads

Lower priority threads won't run unless higher priority threads block (i.e. pause)

- × Requires "good citizen" threads
- × Low priority threads may starve
- ✓ Lower priority threads never break high priority threads
- ✓ Lower context-switch overhead

- **Time-sliced threads have a "nieceness" value by which administrator may modify relative loading**
- **Linux dynamically modifies processes' time slice according to process behavior**
- **With realtime threads, the highest priority thread always runs until it blocks itself**

The Usefulness of Processes

Option 1: Audio and Video in a single Process

```
// audio_video.c
// handles audio and video in
//     a single application

int main(int argc, char *argv[])
{
      while(condition == TRUE){
            callAudioFxn();
            callVideoFxn();
      }
}
```

Option 2: Audio and Video in separate Processes

```
// audio.c, handles audio only

int main(int argc, char *argv[]) {
      while(condition == TRUE)
            callAudioFxn();
}
```

```
// video.c, handles video only

int main(int argc, char *argv[]) {
      while(condition == TRUE)
            callVideoFxn();
}
```

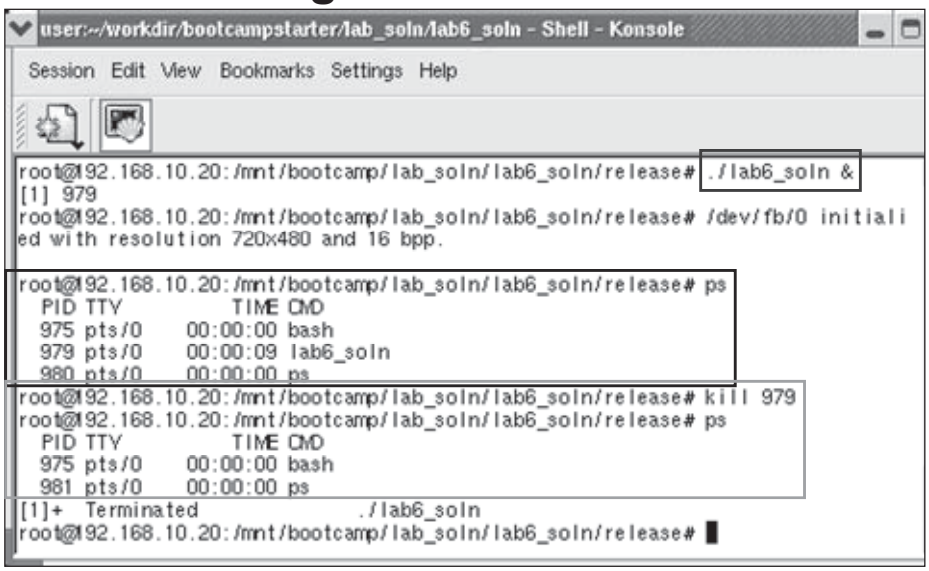
Splitting into two processes is helpful if:

1. audio and video occur at different rates
2. audio and video should be prioritized differently
3. multiple channels of audio or video might be required (modularity)
4. memory protection between audio and video is desired

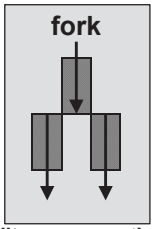
Terminal Commands for Processes

`# ps`	**Lists currently running user processes**
`# ps -e`	**Lists all processes**
`# top`	**Ranks processes in order of CPU usage**
`# kill <pid>`	**Ends a running process**
`# renice +5 –p <pid>`	**Changes time-slice ranking of a process** (range +/- 20)

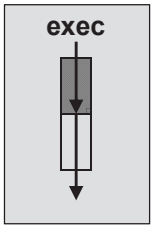
Launching a Process – Terminal



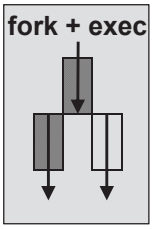
Side Topic – Creating New Processes in C



Splits one executing process into two with same environment



New process replaces old but maintains previous environment

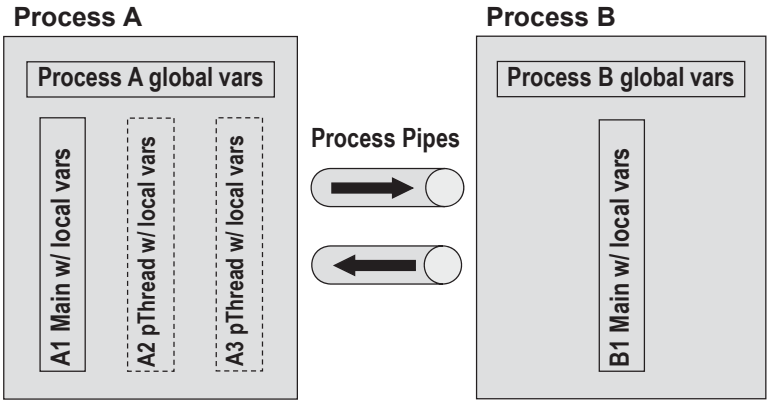


Launch new process and keep previous process

- All processes are *split-off* from the original process created at startup
- When using fork, both processes run the same code; to prevent this, test if newly created process and run another program – or exec to another program
- To review, a *process* consists of:
  - Context (memory space, file descriptors)
  - One (or more) threads

Linux Threads

Processes and Threads

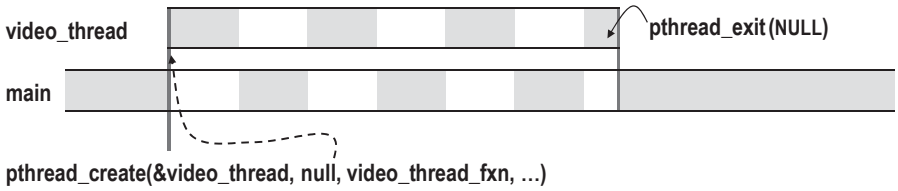


- By default, each process contains one main thread of execution
  - Additional threads can be spawned within a process (pThreads)
  - All threads within a process share global variables
- Threads scheduled individually by priority – regardless of which process they reside within
- No thread isolation – a rogue pointer will probably bring down all threads in that process.

Threads vs Processes

	Processes	Threads
Memory protection	✓	✘
Ease of use	✓	✘
Start-up cycles	✘	✓
Context switch	✘	✓
Shared globals	no	yes
Scheduled entity	no	yes
Environment	program	function

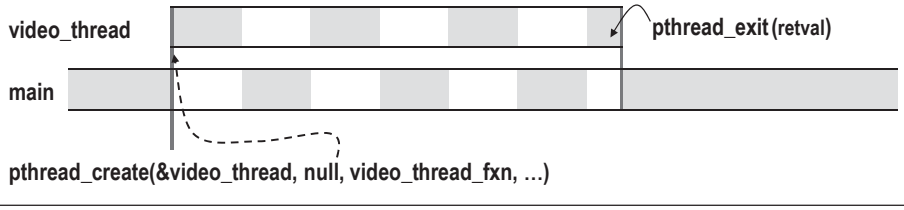
pThread Functions – Create & Exit

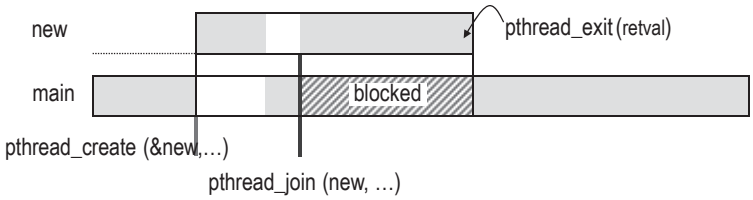



```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
      void *(*start_routine)(void *), void *arg);
pthread_join(pthread_t thread, void **retval);
pthread_exit(void *retval);
```

What if there's nothing for main() to do?

Re-Joining Main





Thread Synchronization

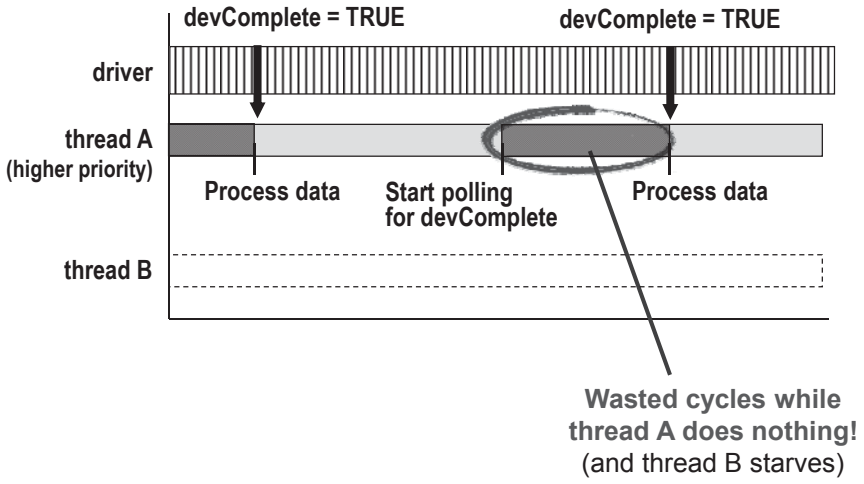
Thread Synchronization (Polling)

```
void *threadA(void *env){
int test;
while(1){
     while(test != TRUE) {
          test = (volatile int) env->driverComplete;
     }
     doSomething(env->bufferPtr);
}
}
```

Polling (spin loop)

- Thread A's doSomething( ) function should only run after the driver completes reading in a new buffer
- Polling can be used to halt the thread in a spin loop until the driverComplete flag is thrown.
- But polling is inefficient because it wastes CPU cycles while the thread does nothing.

Thread Synchronization (Polling)



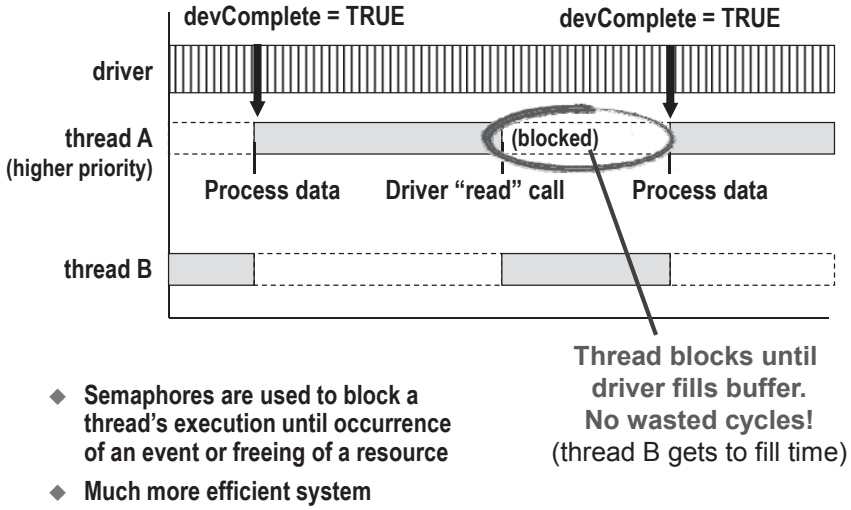
Thread Synchronization (Blocking)

```
void *threadA(void *env){
while(1){
    read(env->audioFd, env->bufferPtr, env->bufsize);
    doSomethingNext(env->bufferPtr);
}
```

Blocking (waits till complete)

- Instead of polling on a flag, the thread blocks execution as a result of the driver's read call
- More efficient than polling because thread A doesn't waste cycles waiting on the driver to fill the buffer

Thread Synchronization (Blocking)



- Semaphores are used to block a thread's execution until occurrence of an event or freeing of a resource
- Much more efficient system

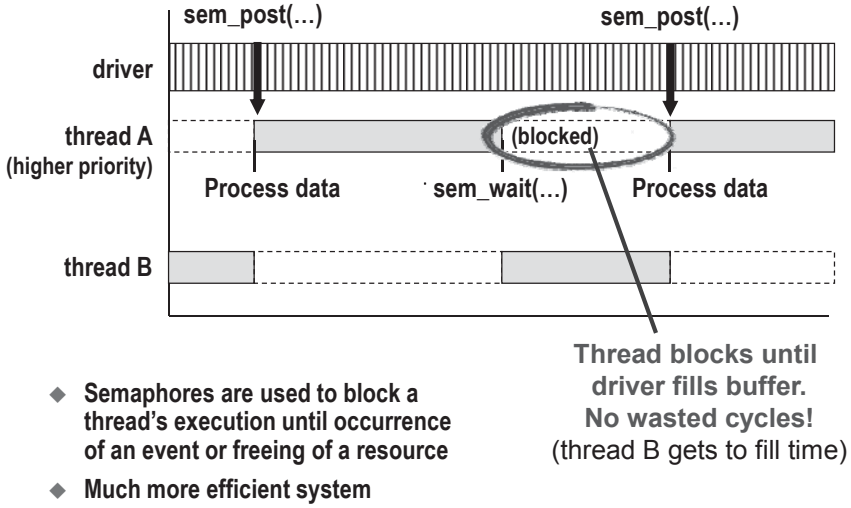
Semaphores

```
#include <semaphore.h>
void *threadA(void *env){
sem_t mySem;
sem_init(&mySem, 1, 0);  // initial value of zero
while(1){
    sem_wait(&mySem);
    doSomethingNext(env->bufferPtr);
}
}
```

Blocking (waits till complete)

- A semaphore is the underlying mechanism of the read call that causes it to block

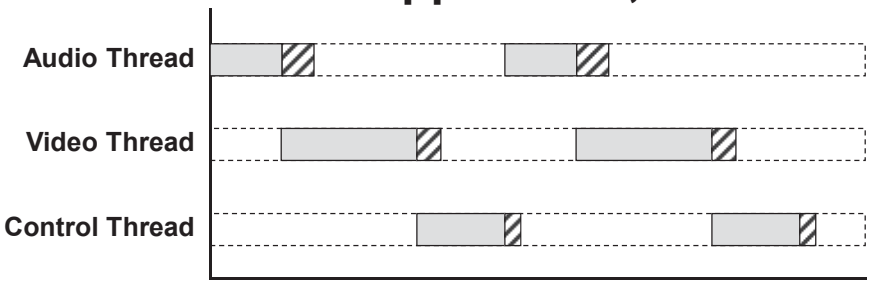
Thread Synchronization (Semaphore)



- Semaphores are used to block a thread's execution until occurrence of an event or freeing of a resource
- Much more efficient system

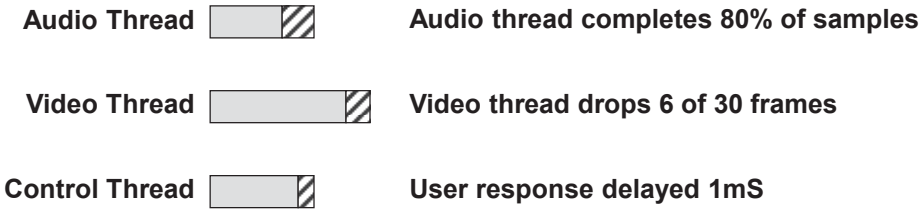
Using Real-Time Threads

Time-Sliced A/V Application, >100% load



- Adding a new thread of the highest "niceness" (smallest time slice) may disrupt lower "niceness" threads (higher time slices)
- All threads share the pain of overloading, no thread has time to complete all of its processing
- Niceness values may be reconfigured, but system unpredictability will often cause future problems
- In general, what happens when your system reaches 100% loading? Will it degrade in a well planned way? What can you do about it?

Time-Sliced A/V Application Analysis

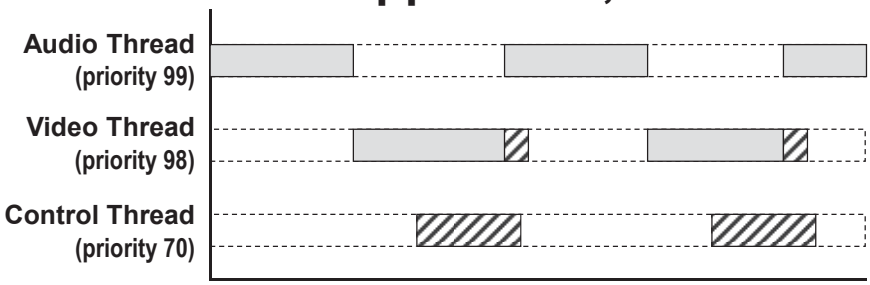


All threads suffer, but not equally:

- Audio thread real-time failure is highly perceptible
- Video thread failure is slightly perceptible
- Control thread failure is not remotely perceptible

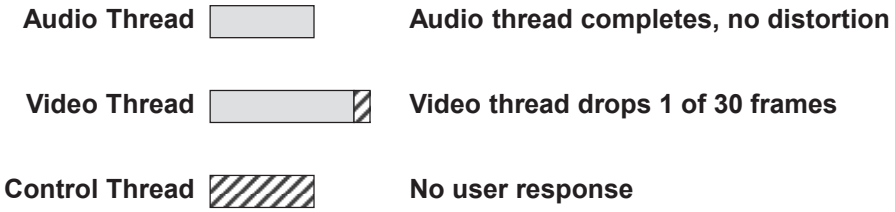
Note:
Time-slicing may also cause real-time failure in systems that are <100% loaded due to increased thread latency

Real-time A/V Application, >100% load



- Audio thread is guaranteed the bandwidth it needs
- Video thread takes the rest
- Control thread never runs!

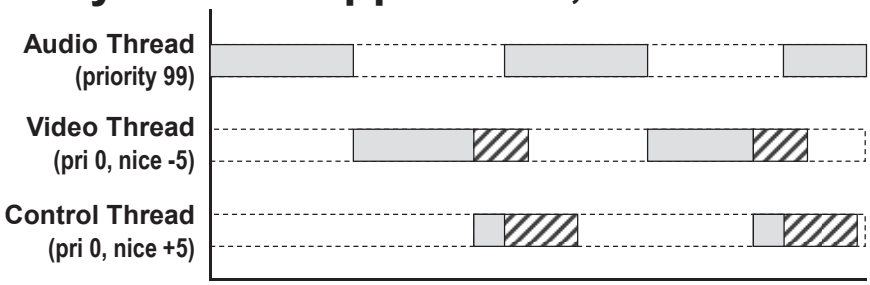
Time-Sliced A/V Application Analysis



Still a problem:

- Audio thread completes as desired
- Video thread failure is practically inperceptible
- Control thread never runs – User input is locked out

Hybrid A/V Application, >100% load



- Audio thread is guaranteed the bandwidth it needs
- Video thread takes *most* of remaining bandwidth
- Control thread gets a small portion of remaining bandwidth

Hybrid A/V Application Analysis

Audio Thread		Audio thread completes, no distortion
Video Thread		Video thread drops 2 of 30 frames
Control Thread		User response delayed 100ms

A good compromise:

- Audio thread completes as desired
- Video thread failure is barely perceptible
- Control thread delayed response is acceptible
- Bottom Line: We have designed the system so that it degrades gracefully

Default Thread Scheduling

```
#include <pthread.h>
  …
pthread_create(&myThread, NULL, my_fxn,
                                  (void *) &audio_env);
```

- **Setting the second argument to NULL means the pthread is created with default attributes**

pThread attributes:	NULL / default value:
stacksize	PTHREAD_STACK_MIN
…	…
detachedstate	PTHREAD_CREATE_JOINABLE
schedpolicy	**SCHED_OTHER (time slicing)**
inheritsched	PTHREAD_INHERIT_SCHED
schedparam.sched_priority	0

Scheduling Policy Options

	SCHED_OTHER	SCHED_RR	SCHED_FIFO
Sched Method	**Time Slicing**	**Real-Time (RT)**	
RT priority	0	1 to 99	1 to 99
Min niceness	+20	n/a	n/a
Max niceness	−20	n/a	n/a
Scope	**root or user**	**root**	**root**

- Time Sliced scheduling is specified with **SCHED_OTHER**:
  - Niceness determines how much time slice a thread receives, where higher niceness value means less time slice
  - Threads that block frequently are rewarded by Linux with lower niceness
- Real-time threads use preemptive (i.e. priority-based) scheduling
  - Higher priority threads always preempt lower priority threads
  - RT threads scheduled at the same priority are defined by their policy:
    - **SCHED_FIFO**: When it begins running, it will continue until it blocks
    - **SCHED_RR**: "Round-Robin" will share with other threads at it's priority based on a deterministic time quantum

Real-time Thread Creation Procedure

Create attribute structure

```
// Initialize the pthread_attr_t structure audioThreadAttrs
pthread_attr_init(&audioThreadAttrs);
```

Set attribute to real-time priority 99

```
// Set the inheritance value in audioThreadAttrs structure
pthread_attr_setinheritsched(&audioThreadAttrs,
                                        PTHREAD_EXPLICIT_SCHED);

// Set the scheduling policy for audioThreadAttrs structure
pthread_attr_setschedpolicy(&audioThreadAttrs, SCHED_RR);

// Set the scheduler priority via audioThreadParams struct
audioThreadParams.sched_priority = 99;
pthread_attr_setschedparam(&audioThreadAttrs,
                                        &audioThreadParams);
```

Create thread with given attributes

```
// Create the new thread using thread attributes
pthread_create(&audioThread, &audioThreadAttrs,
                        audio_thread_fxn, (void *) &audio_env);
```

Linking the pthread Library

- **pthread stands for POSIX thread**
- **POSIX threads were introduced to Linux in 1996, but are not widely used because of the (relatively) low overhead of using processes.**
- **What does POSIX mean?**
  - **IEEE POSIX committee defined POSIX 1003.1c standard for threads.**
  - **Linux threads are modeled after this standard, though don't completely adhere to it.**

#include <pthread.h>

#include	C compiler pre-processor statement
pthread.h	Function declarations for the posix thread library

 # gcc –D_REENTRANT myProg.c –o myprog.o –lpthread

gcc	GNU compiler collection. Invokes compiler, assembler and linker as necessary for specified source files.
-D_REENTRANT	Defines the _REENTRANT symbol, which forces all included libraries to use reentrant functions.
-lpthread	Link in the pthread (POSIX thread) library.

(Page intentionally left blank)

Lab 3: 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.

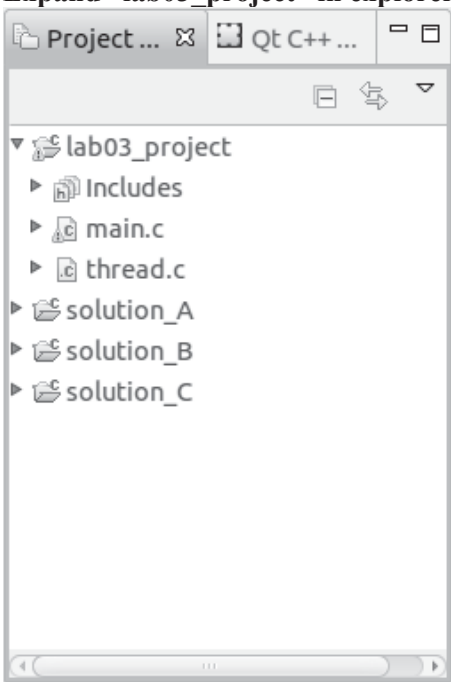
Module Topics

A.Creating a POSIX thread

1. **If needed, start Code Composer Studio from the desktop icon**
2. **Switch to the "lab03_workspace" workspace**
   File→Switch Workspace

   The workspace is located at `/home/user/labs/lab03_workspace`

3. **Expand "lab03_project" in explorer view**



These are the starting files for the project. There is a small exclaimation mark next to main.c because

You have been provided with three starting files. "thread.c" and "thread.h" are helper files that you should examine but do not need to modify. "main.c" is basically empty and will be where you write your application.

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**

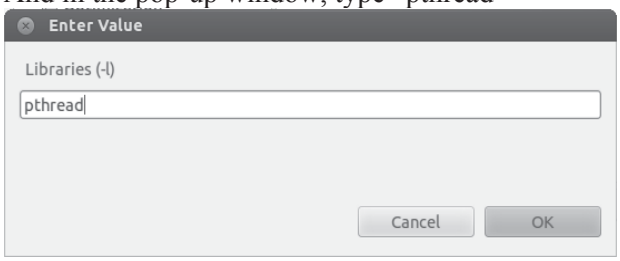
```
/* Global thread environments */
typedef struct thread_env
{
    int quit;     // Thread will run as long as quit = 0
    int id;
    sem_t *mySemPtr;
    sem_t *partnerSemPtr;
} thread_env;

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.

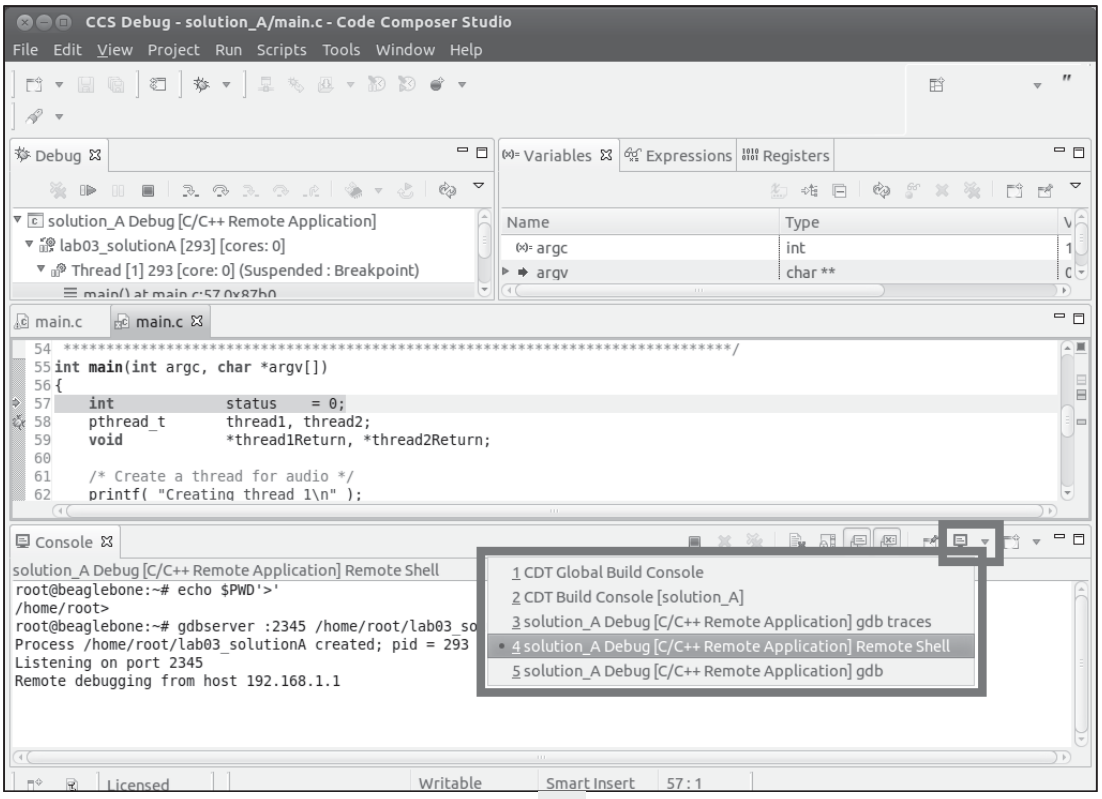
7. **Write the thread function**
   For this first stage of the lab, you will only use the "quit" and "id" fields of the environment structure. Your thread function should have three phases:
   1. Print a message to stdout (will be printed to terminal) indicating that the thread has been entered. Be 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. **Link the pthread library into the project**
   (Right-click lab03_project)→Properties…
   C/C++ Build→Settings→Cross G++ Linker→Libraries
   Press the add library button

   And in the pop-up window, type “pthread”



10. **Build lab03_project**
    In the project explorer, (right-click lab03_project)→Build Project
    or
    Project→Build All (ctrl-B)
11. **Once the program has built, launch a debug session**
    Use “run→Debug Configurations…” and select “lab03_project Debug”
    You should be taken to main
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

```
root@beaglebone:~# echo $PWD'>'
/home/root>
root@beaglebone:~# gdbserver :2345
/home/root/lab03_solutionA;exit
Process /home/root/lab03_solutionA created; pid = 293
Listening on port 2345
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 #2
Inside while loop of 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
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 #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:

```
root@beaglebone:~# echo $PWD'>'
/home/root>
root@beaglebone:~# gdbserver :2345
/home/root/lab03_solutionB;exit
Process /home/root/lab03_solutionB created; pid = 303
Listening on port 2345
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
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
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 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."
    The prototype for sem_init( ) is:

    ```
    int sem_init(sem_t *sem, int pshared, unsigned int value);
    ```

    The function returns 0 if successful, negative if failure.
    "sem" is a semaphore handle passed by reference.
    "pshared" is 0 if local, 1 if global. Either can be used.
    "value" is the initial value of the semaphore, which should be initialized 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:

```
root@beaglebone:~# echo $PWD'>'
/home/root>
root@beaglebone:~# gdbserver :2345
/home/root/lab03_solutionC;exit
Process /home/root/lab03_solutionC created; pid = 313
Listening on port 2345
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
```

Module 04: Linux File and Driver I/O

Introduction

A basic feature of nearly any modern operating system is the abstraction of peripherals via device drivers. Linux provides a basic open-read-write-close interface to peripheral drivers as well as standard files.

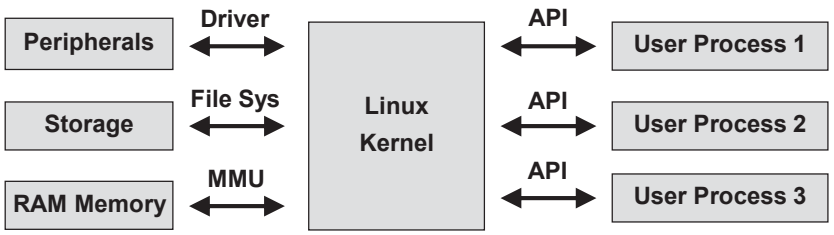
The Linux operating system has a wide range of filesystem support for many different block storage devices. The ability to store and manipulate files on NAND and NOR flash, multimedia and secure digital cards, hard disk drives and other media is important for many systems.

This section begins with an overview of Linux drivers that is applicable both to storage media drivers and device drivers, then provides a brief discussion on the differences between the basic driver model and the file I/O model. It finishes with the case study of a special driver: sending Ethernet messages via Berkeley Sockets.

Module Topics

Driver Basics

Kernel vs User Space



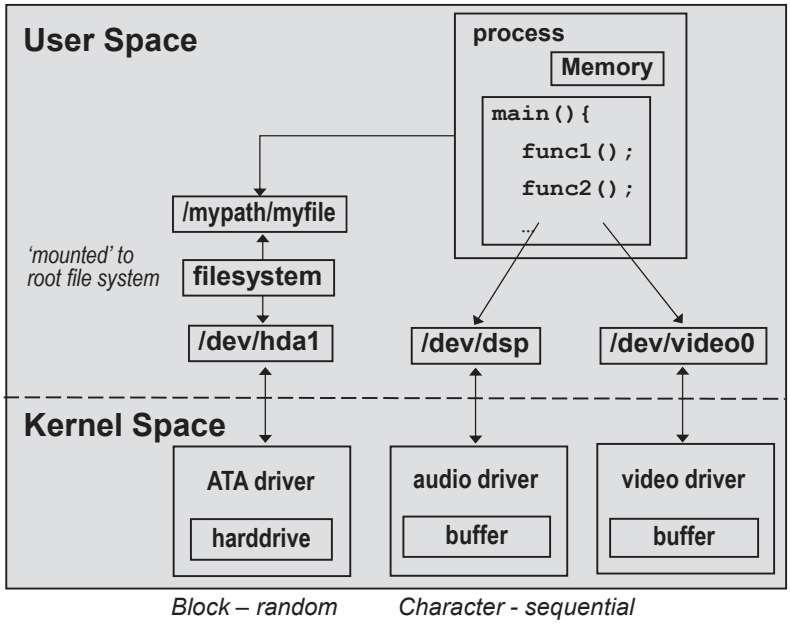
Kernel Space

- The Linux kernel manages the machine's hardware via drivers
- Drivers provide hardware abstraction and protection
- Drivers execute within the Linux kernel's context, and are inserted into the running Linux kernel or statically linked in when kernel is built

User Space(Applications)

- Cannot access peripherals or memory directly
- Make requests to kernel to access peripherals or storage media

User Access to Kernel Space



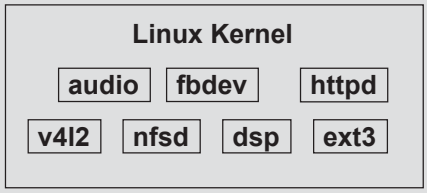
Four Steps to Accessing Drivers

1. **Load the driver's code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (block drivers only)**
4. **Access resources using open, read, write and close**

Kernel Object Modules

How to add modules to Linux Kernel:

1. Static (built-in)



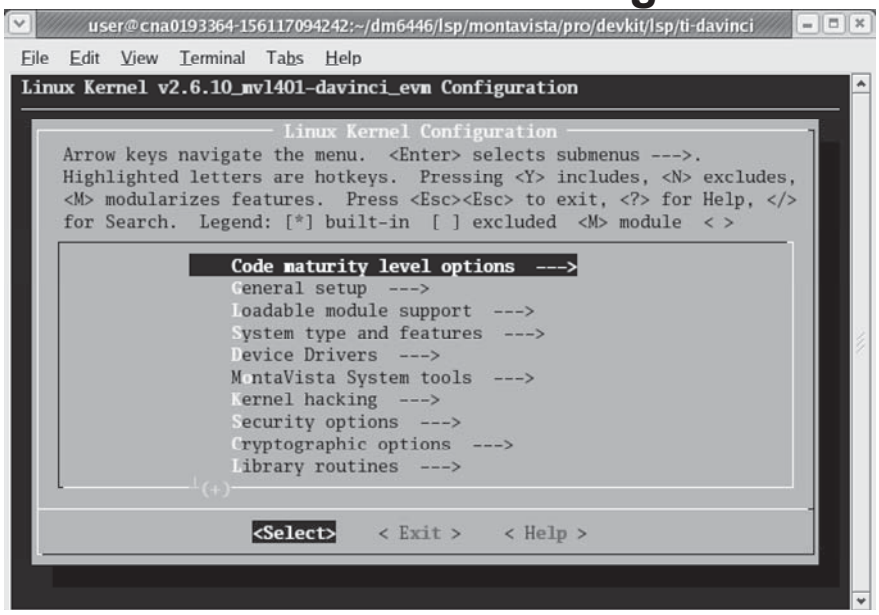
Kernel Module Examples:

fbdev	frame buffer dev
v4l2	video for linux 2
nfsd	network file server dev
dsp	oss digital sound proc.
audio	alsa audio driver

- **Linux Kernel source code is broken into individual modules**
- **Only those parts of the kernel that are needed are built in**

Change static configuration using...

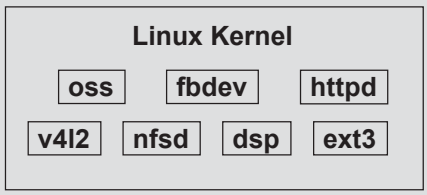
Static Linux Kernel Configuration



Kernel Object Modules

How to add modules to Linux Kernel:

1. Static (built-in)



Kernel Module Examples:

fbdev	frame buffer dev
v4l2	video for linux 2
nfsd	network file server dev
dsp	oss digital sound proc.
audio	alsa audio driver

- **Linux Kernel source code is broken into individual modules**
- **Only those parts of the kernel that are needed are built in**

2. Dynamic (modprobe)

```
# modprobe <mod_name>.ko [mod_properties]
```

- **Use modprobe command to dynamically add modules into the Linux kernel**
- **Keep statically built kernel small (to reduce size or boot-up time), then add functionality later with modprobe**
- **modprobe is also handy when developing kernel modules**

.ko = kernel object

Examining The Steps in More Detail…

1. **Load the driver's code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (block drivers only)**
4. **Access resources using open, read, write and close**

Linux Driver Registration

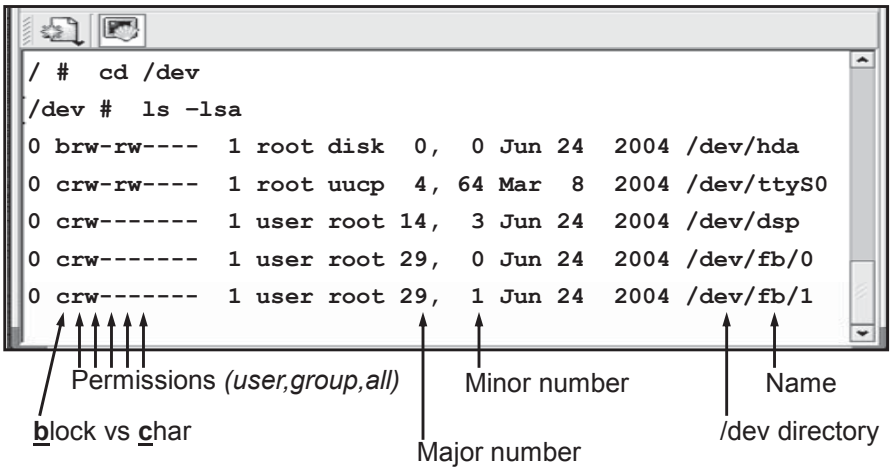
```
# mknod <name> <type> <major> <minor>

   <name>:  Node name (i.e. virtual file name)
   <type>:  b    block
            c    character
  <major>:  Major number for the driver
  <minor>:  Minor number for the driver
```

```
Example:    mknod  /dev/fb/3  c  29  3

Useage:     Fd = open("/dev/fb/3", O_RDWR);
```

- **Register new drivers with *mknod* (i.e. Make Node) command.**
- **Major number** determines which driver is used (the name does not affect which driver is used). Most devices have number assigned by Linux community.
- **Minor number** is significant for some drivers; it could denote instance of given driver, or in our example, it refers to a specific buffer in the FBdev driver.

Linux Device Registration

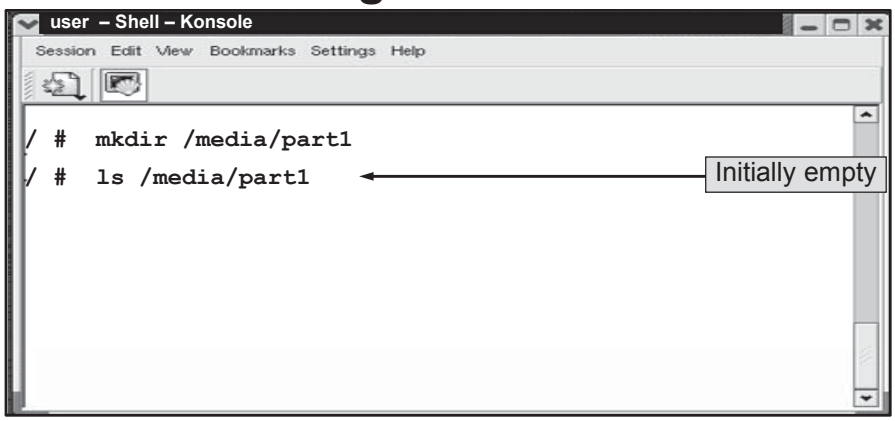
- Linux devices are registered in /dev directory
- Two ways to view registered devices:
  - `cat /proc/devices`
  - `ls -lsa` command `(as shown below)` to list available drivers



Examining The Steps in More Detail…

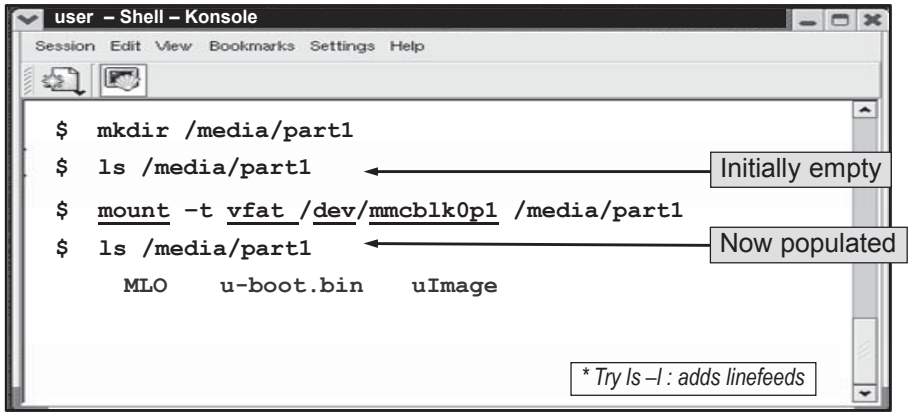
1. Load the driver's code into the kernel (insmod or static)
2. Create a virtual file to reference the driver using mknod
3. Mount block drivers using a filesystem (block drivers only)
4. Access resources using open, read, write and close

Mounting Block Devices



- Mounting a block driver into the filesystem gives access to the files on the device as a new directory
- Easy manipulation of flash, hard drive, compact flash and other storage media
- Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem

Mounting Block Devices



- Unlike Windows, there is only one filesystem – therefore you must mount to a mount point (i.e. empty directory) in the root filesystem
- Easy manipulation of flash, hard drive, compact flash and other storage media
- Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem
- The above example shows mounting an external harddrive into the root filesystem

Some Common Filesystem Types

Harddrive File systems:

ext2	Common general-purpose filesystem
ext3	Journaled filesystem - Similar to ext2, but more robust against unexpected power-down
vfat	Windows “File Allocation Table” filesystem

Memory File systems:

jffs2	Journaling flash filesystem (NOR flash)
yaffs	yet another flash filesystem (NAND flash)
ramfs	Filesystem for RAM
cramfs	Compressed RAM filesystem

Network File systems:

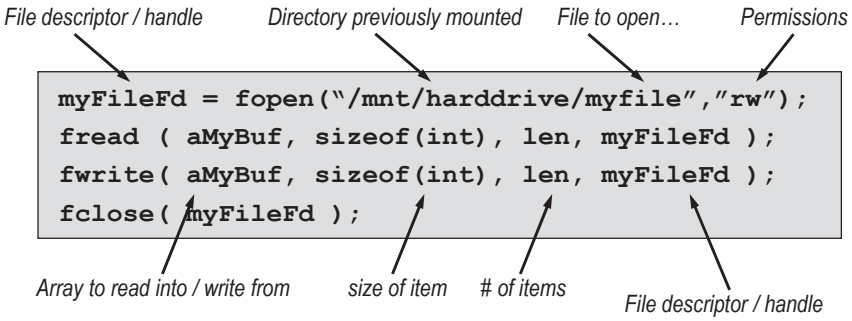
nfs	Share a remote linux filesystem
smbfs	Share a remote Windows® filesystem

Examining The Steps in More Detail…

1. **Load the driver’s code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (block drivers only)**
4. **Access resources using open, read, write and close**

Accessing Files

Manipulating files from within user programs is as simple as…



Additionally, use fprintf and fscanf for more feature-rich file read and write capability

Using Character Device Drivers

Simple drivers use the same format as files:

```
soundFd = open("/dev/dsp", O_RDWR);
read ( soundFd, aMyBuf, len );
write( soundFd, aMyBuf, len );
close( soundFd );
```

Additionally, drivers use I/O control (ioctl) commands to set driver characteristics

```
ioctl( soundFd, SNDCTL_DSP_SETFMT, &format);
```

Notes:

- len field is always in bytes
- More complex drivers, such as V4L2 and FBDEV video drivers, have special requirements and typically use ioctl commands to perfrom reads and writes

Ethernet Basics

Networking References

`FF.FF.FF.FF.FF.FF` — Media Access Controller address is unique to a physical device

`255.255.255.255` — IP address is assigned by network or network administrator

`www.ti.com` — Host name and domain name are resolved by domain name servers

Address Resolution Protocol (ARP) resolves MAC addresses from IP addresses

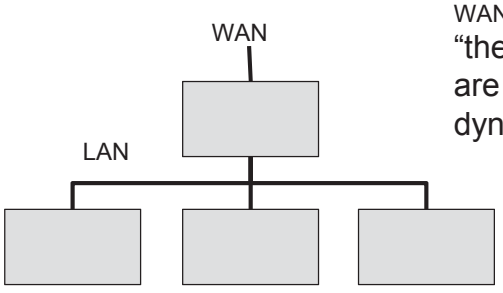
Domain Name Service (DNS) resolves IP addresses from host and domain name

Sub-networks

`FF.FF.FF.FF.FF.FF`

`255.255.255.255`

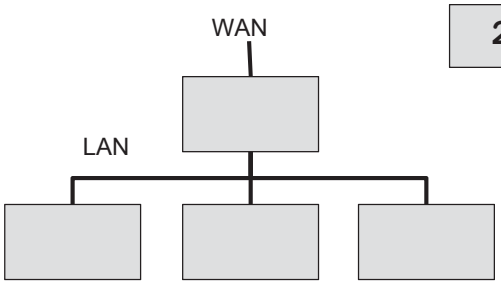
2^48 MAC Addresses
>>
2^32 IP Addresses



WAN = Wide Area Network
"the cloud." WAN addresses are unique (but may be dynamic)

LAN = Local Area Network (subnet.)
IP addresses are not unique

Sub-networks



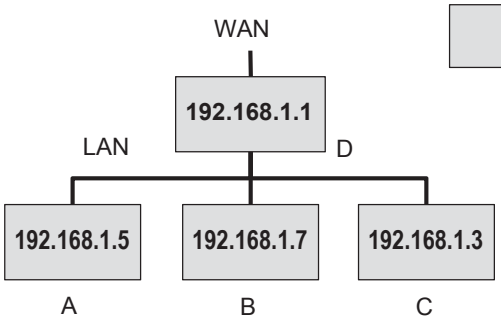
255.255.255.255

Sub-networks are formed using a netmask

If <Dest> & Netmask = <Source> & Netmask then source and destination are on the same subnet.

Else, destination packet is forwarded to the WAN.

Sub-network Example

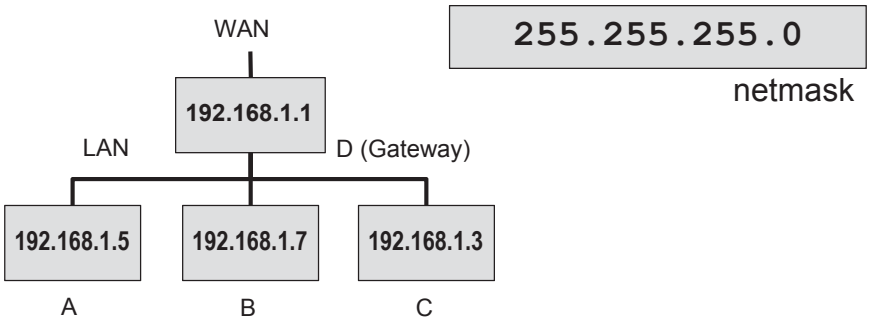


255.255.255.0

netmask

Computer A sends a packet to IP address 192.168.1.7
Computer A (192.168.1.5) & 255.255.255.0 = 192.168.1.0
Destination & 255.255.255.0 = 192.168.1.0
This is a transmission within the subnet, packet is sent directly to destination.

Sub-network Example



Computer A sends a packet to IP address 116.97.23.47
Computer A (192.168.1.5) & 255.255.255.0 = 192.168.1.0
Destination & 255.255.255.0 = 116.97.23.0
Packet is forwarded to the Gateway to be sent to the wide area network.

IP Ports

In addition to an IP address, connection requests may specify a port number. Ports are generally used in order to route connection requests to the appropriate server.

Service	Port
echo	7
ftp	20/21
ssh	22
telnet	23
nameserver	42
http	80

Client/Server model

Server is a static background (daemon) process, usually initiated at startup, that listens for requests from one or more clients.

Berkeley Sockets

Berkeley Sockets Basic Procedure

Client

```
sFD = socket(…);


connect(sFD, addr, …);
// do reads and writes
shutdown(sFD,…);
close(sFD);
```

Server

```
sFD = socket(…);
bind(sFD, addr, …);
listen(sFD, …);
while(1){
    cFD = accept(sFD,…);
    // do reads and writes
    shutdown(cFD,…);
    close(cFD);
}
```

Socket Types

Local Socket

```
sFD = socket(AF_UNIX, SOCK_STREAM, 0);
```

TCP/IP Socket

```
sFD = socket(AF_INET, SOCK_STREAM, 0);
```

UDP/IP Socket

```
sFD = socket(AF_INET, SOCK_DGRAM, 0);
```

Final parameter to socket function is protocol. Protocol is typically determined completely by first two parameters, so that “0” (default) is usually passed.

Specifying Address and Port

```
struct sockaddr_in stSockAddr;

memset(&stSockAddr, 0, sizeof(stSockAddr));

stSockAddr.sin_family = AF_INET;
stSockAddr.sin_addr.s_addr = inet_addr("192.168.1.1");
stSockAddr.sin_port = htons(1100);
```

Client

```
connect(SocketFD, (struct sockaddr *)&stSockAddr,
       sizeof(stSockAddr));
```

Server

```
bind(SocketFD,(struct sockaddr *)&stSockAddr,
       sizeof(stSockAddr))
```

Server Details

listen(int sFD, int backlog);

listen creates an incoming connection queue

socketFD is the file descriptor

backlog is the maximum number of connection requests to hold in the queue

accept(int sFD, sockaddr *address, size_t *len);

address and len (of address struct) are used to return the address of the connecting client. If this is not of interest, NULL may be passed.

shutdown**(int sFD, int how);**

how may be SHUT_RD, SHUT_WR, SHUT_RDWR

possible to call with SHUT_RD followed by SHUT_WR

shutdown informs TCP/IP stack to terminate session, but does not close file descriptor. Many systems implement shutdown automatically in close, but best to always call.

(Page intentionally left blank)

Lab 4: Berkeley Sockets

Introduction

In this lab exercise you will explore Linux networking on the Beaglebone Black. You will write a Berkeley sockets client application which will send the message “Hello World!” from the Beaglebone 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 Beaglebone are shown preceded with the beaglebone prompt:

```
beaglebone$
```

Module Topics

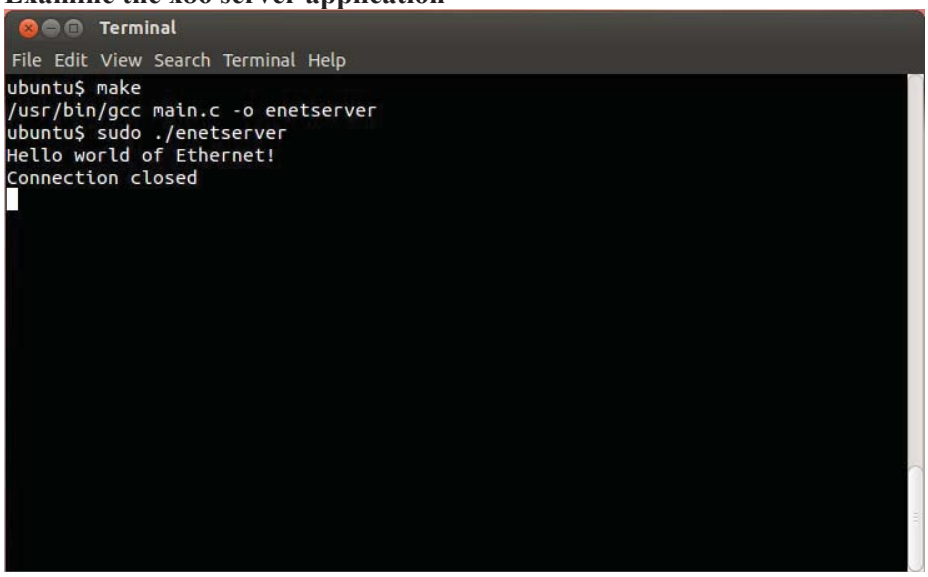
A. Berkeley Socket Client

1. **Start code composer studio and switch to "/home/user/labs/lab04_workspace"**
   File→Switch Workspace
2. **Examine main.c from lab04_project**
   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 <u>do not</u> 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/labs/lab04solution_x86_server` directory**
   Be careful not to change into the "lab04_workspacet" or "lab04_x86_server" directories by mistake.
10. **List the contents of the directory**
    There are two files: makefile and main.c
11. **Rebuild the application**
    ```
    ubuntu$ make
    ```
    After a successful completion of the make script, a new executable named "enetserver" will appear in the directory.
12. **Launch the application with root permissions using the "sudo" command**
    ```
    ubuntu$ sudo ./enetserver
    ```
    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 lab04_project 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**



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.

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 "/home/user/labs/lab04_x86_server" directory**
18. **Open main.c for editing**
    ```
    ubuntu$ gedit main.c
    ```
19. **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.
20. **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)

21. **Ready the socket to accept connections with the "listen" command**
22. **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.
23. **Save the file**
24. **Rebuild the server executable**
    ```
    ubuntu$ make
    ```
25. **Test your server executable with the known-good client that you wrote in section C**

D.(Optional) Challenge

As an optional challenge, see if you can modify the server and client applications to use the Ethernet-over-USB connection instead of the gigabit Ethernet connection. These use the addresses:

X86 Host: 192.168.2.1

Beaglebone: 192.168.2.2

If you also create a new CCS connection using Ethernet over USB and modify the debug configuration to use this connection, you can completely remove the Ethernet cable to test the application.

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 four 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

Chapter Topics

Installing Ubuntu 12.04

There are many tutorials available for installing Ubuntu, so this section will not go through great detail on the actual installation; however, it provides an optional section 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.**
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 3.
   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.5.3.0.00090

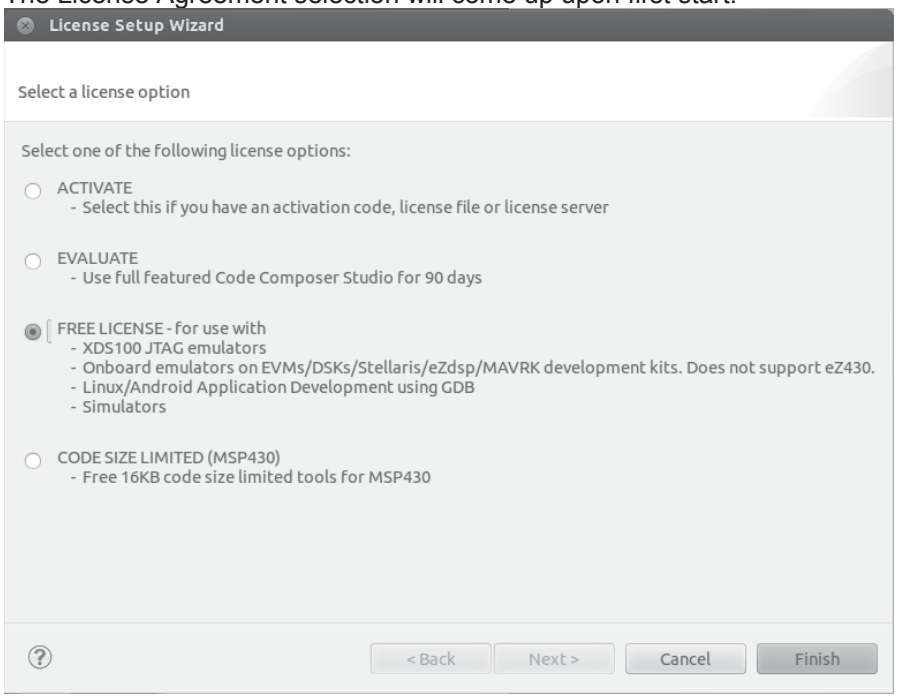
You can download CCS from:

`http://processors.wiki.ti.com/index.php/Download_CCS`

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. **Download CCS from the above listed link**
10. **Add executable permission to the installer**
    `# chmod a+x ccs_setup_5.4.0.00090.bin`
11. **Run the installer program**
    `# ./ccs_setup_5.4.0.00090.bin`
    Note: do not run the installer with root (i.e. sudo) permissions.
    The installer will give you a message that Emulation drivers can only be installed with root permissions, but we will not be using emulation drivers, so this is not an issue.
12. **Read and accept the license agreement.**
13. **Accept the default install directory: /home/user/ti**
14. **Choose custom install**
15. **Select “AMxx Cortex-A and ARM9 processor” from the Processor Support window**
16. **In “Select Components” window, choose:**
    Compiler Tools → GCC ARM Compiler Tools
    And deselect the other choices
17. **Deselect all emulator support except “TI Emulators” on the next screen**
    The workshop does not use “TI Emulators” but there is no way to deselect in the installer.
18. **Press “Next” on the next two screens to begin the installation**
19. **Select “Create Desktop Shortcut” on final screen (Should be selected by default.)**
20. **Start Code Composer Studio**
21. **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…

22. **Close CCS**

Installing Lab Files

Note that the Code Composer Studio project files that have been set up for the workshop expect for the labs folder to be extracted into the "/home/user" directory. If you have a previous Ubuntu installation with a different user name, it is recommended that you create a new user named "user" and install the lab files into this user's directory. You should then run the lab exercises when logged in as "user."

23. **Update the Aptitude package manager**
    ubuntu$ sudo apt-get update
24. **Install the sg3-utils package**
    ubuntu$ sudo apt-get install sg3-utils
25. **Download lab files**
    ubuntu$ firefox
    browse to:
    http://processors.wiki.ti.com/index.php/Introduction_to_Linux_One-Day_Workshop
    Download "bbb_linux_oneday_labs.tar.gz"
26. **Install lab files**
    ```
    ubuntu$ tar -zxf bbb_linux_oneday_labs.tar.gz -C /home/user
    ```

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”

27. **Open /etc/network/interfaces file**
    # sudo gedit /etc/network/interfaces
28. **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 192.168.2.1
      netmask 255.255.255.0
```

    Note: “address” and “netmask” entries preceded by tab.
29. **Save and save and close**
30. **(Optional) Reboot and use “ifconfig” to verify new setting**

```
# ifconfig
```

31. **Create a file /lib/udev/rules.d/96-usb0.rules**

```
# sudo gedit /lib/udev/rules.d/96-usb0.rules
```

32. **Remove Gnome networking settings**

```
# sudo nm-connection-editor
```

    Any connection that appears under the “wired” or “wireless” tab should be deleted.
33. **Open /etc/hosts**

```
# sudo gedit /etc/hosts
```

34. **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 beaglebone.gigether.net
192.168.2.1 ubuntu.etherusb.net
192.168.2.2 beaglebone.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.

35. **Launch a terminal**
36. **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
    ```
37. **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
    ```
38. **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.
39. **Select the Gnome Desktop**
    Click the Ubuntu icon next to thye username (user) and select Gnome.
40. **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.)
41. **Disable overylay scrollbars**
    Launch a terminal and type the following (single line, no carriage return)
    ```
    ubuntu$ gsettings set org.gnome.desktop.interface ubuntu-overlay-scrollbars false
    ```
42. **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.

(Optional) Installing Angstrom Cross-compile Tools

NOTE: All of the cross-compile tools built in this section are included in the workshop lab installation file, and do not need to be rebuilt in order to run the workshop. These steps are included here as a reference for those who wish to know what was done.

The Angstrom distribution that is used in the workshop provides a set of cross-compile tools. These can be rebuilt using the bitbake build system of OpenEmbedded.

Note that in step 52, there is an install script generated at ~/Anstrom/setup-scripts/build/tmp-angstrom_v2012_12-eglibc/deploy/angstrom-eglibc-i686-armv7a-vfp-neon-v2012.12-toolchain.sh; however, this install script contains a wrapper that requires executing "`. ./<install_dir>/environment-setup-armv7a-vfp-neon-anstrom-linux-gnueabi`" and then compiling using "`$CC main.c`"

This wrapper is needed for compatibility with the makefile structure used in Texas Instruments Software Development Kits (SDKs), but adds unnecessary extra build steps when the SDK is not used, as in this workshop. Thus, it was decided to copy the raw, unwrapped tools directly out of the Bitbake build directories.

43. **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
    ```
44. **Install Bitbake/OE dependencies**
    ```
    ubuntu$ sudo apt-get update
    ubuntu$ sudo apt-get install subversion cvs git-core \
    build-essential help2man diffstat texi2html texinfo \
    libncurses5-dev gawk python-dev python-pysqlite2 \
    gnome-doc-utils gettext automake flex chrpath
    ```
45. **Install git**
    ```
    ubuntu$ sudo apt-get install git
    ```
46. **"git" Angstrom install scripts**
    ```
    ubuntu$ cd ~
    ubuntu$ mkdir Angstrom
    ubuntu$ cd Angstrom
    ubuntu$ git clone git://github.com/Angstrom-distribution/setup-scripts.git
    ```
47. **Link /bin/sh into /bin/bash (currently it is /bin/dash)**
    The OpenEmbedded build system requires /bin/bash
    ```
    ubuntu$ sudo rm /bin/sh
    ubuntu$ sudo ln –s /bin/bash /bin/sh
    ```
48. **Configure OpenEmbedded for beaglebone**
    ```
    ubuntu$ cd setup-scripts
    ubuntu$ MACHINE=beaglebone ./oebb.sh config beaglebone
    ```
49. **Build the cross-compiler toolchain**
    ```
    ubuntu$ MACHINE=beaglebone ./oebb.sh bitbake meta-toolchain
    ```
50. **Build cross gdb (the gnu debugger)**
    ```
    ubuntu$ MACHINE=beaglebone ./oebb.sh bitbake gdb-cross
    ```
51. **Make toolchain directory**
    ```
    ubuntu$ mkdir ~/labs/lab00_toolchain
    ```

52. **Copy cross toolchain and cross gdb**
    ```
    ubuntu$ cp –R ~/Angstrom/setup-scripts/build/tm-
    angstrom_v2012_12-eglibc/sysroots/i686-linux
    ~/labs/lab00_toolchain
    ```
53. **Build gdbserver (for Beaglebone, part of gdb package for Beaglebone)**
    ubuntu$ MACHINE=beaglebone ./oebb.sh bitbake gdb
54. **Insert Beaglebone Black micro-SD card into cardreader and host machine**
55. **Copy Angstrom sysroot to Beaglebone micro-SD card**
    ```
    ubuntu$ sudo cp –R /home/user/Angstrom/setup-scripts/build/tmp-
    angstrom_v2012_12-eglibc/sysroots/beaglebone/* /media/Angstrom
    ```
56. **Copy gdbserver to Beaglebone micro-SD card**
    ```
    ubuntu$ sudo cp /home/usr/Angstrom/setup-scripts/build/tmp-
    angstrom_v2012_12-eglibc/work/armv7a-vfp-neon-angstrom-linux-
    gnueabi/gdb-7.5-r0.0/package/usr/bin/gdbserver
    /media/Angstrom/usr/bin/gdbserver
    ```
57. **Eject micro-SD**
    ```
    ubuntu$ sudo eject /media/Angstrom
    ```

(Optional) Making Cross-Compile Tools Relocatable

The above installation steps will work for building your Cross Compile toolset; however, if you were to remove the “/home/user/Angstrom” directory, the tools would no longer work. This is because they contain a “sysroot” reference into the original build directory which tells the tools where to find common header files and libraries such as the standard C library and associated header file.

As with the other optional sections involving the lab install, this has alredy been done in the `bbb_linux_oneday_labs.tar.gz` archive file, and is only listed here for those wishing to know the steps.

The following steps were taken in the lab00_toolchain files to remove this dependency.

1. **Copy the lab00_targetlibs files into the lab00_toolchain directory**
   This will provide a common path to all needed files
   ```
   ubuntu$ cd /home/user/labs
   ubuntu$ cp –R lab00_targetlibs lab00_toolchain
   ```
2. **Change into the tools directory**
   ```
   ubuntu$ cd usr/bin/armv7a-vfp-neon-angstrom-linux-gnueabi
   ```
3. **Create the addsysroot.sh script**
   ```
   ubuntu$ gedit addsysroot.sh
   ```
4. Place the following into the file (for loop line wraps around in below doc):

   ```
   #!/bin/bash

   for FILE in arm-angstrom-linux-gnueabi-gcc arm-angstrom-linux-
   gnueabi-g++
   do
         mv $FILE $FILE.real
         echo #!/bin/bash > $FILE
         echo DIR=\$\(dirname \$0\) >> $FILE
         echo \$DIR/$FILE.real --sysroot=\$DIR/../../.. \$@ >> $FILE
         chmod a+x $FILE
   done
   ```

5. **Save the script and exit from gedit**
6. **Make the script executable**
   ubuntu$ chmod a+x addsysroot.sh
7. **Run the script**
   ubuntu$ ./addsysroot.sh

(Optional) Modifying Angstrom Filesystem

NOTE: All of the filesystem modifications listed in this section have already been made to the filesystem image included in the lab exercises installation file and do not need to be rebuilt in order to run the workshop. These steps are included here as a reference for those who wish to know what was done.

By default the Angstrom Filesystem used in this workshop uses DHCP to acquire an IP address. This section demonstrates using the "connman" (connection manager) utility from Angstrom in order to set a static IP address.

The ethernet over usb gadget driver cannot be configured using connman, so a systemd startup script is created to launch this driver at each startup. (Angstrom uses systemd instead of sysV as its startup scripting.)

The starting micro-SD card image is:

Angstrom-Cloud9-IDE-GNOME-eglibc-ipk-v2012.12-beaglebone-2013.06.17.img.xz

Which was downloaded from:

http://downloads.angstrom-distribution.org/demo/beaglebone/

The finished filesystem is provided on the workshop wiki page, but the steps are also listed here.

58. **Boot the Beaglebone Black attached to a router that provides access to the wide area network**
59. **Browse to the router, log in and determine the IP address of the Beaglebone Black from the DHCP list of the router.**
    If you are unsure of the IP address of the router, it is usually either 192.168.1.1 or 10.0.0.1
    If neither of these works, attach an x86 PC, acquire a DHCP address from the router, and check the gateway address in your IP settings.
60. **Force Ubuntu to acquire a DHCP address from the router**
    This is necessary assuming that you have set the Ubuntu IP address statically as per the previous section.
    ```
    ubuntu# sudo ifdown eth0
    ubuntu# sudo dhclient eth0
    ```
61. **From Ubuntu, create a secure shell connection to the Beaglebone Black**
    ```
    ubuntu# ssh root@192.168.1.2
    ```
    You may be told that there is no ssh key for the connection and asked if you would like to create one, to which say "yes."
    When you log into the Beaglebone Black, there is no password (press enter when prompted.)
62. **On the Beaglebone Black, update the opkg package manager**
    ```
    bbb# opkg update
    ```
63. **Install connman-tests package**
    ```
    bbb# opkg install connman-tests
    ```
64. **Change to the connman test script directory**
    ```
    bbb# cd /usr/lib/connman/test
    ```
65. **List the currently configured connections**
    ```
    bbb# ./get-services
    ```
    The first line of the output should list something similar to:
    ```
    [ /net/connman/service/ethernet_405fc276b749_cable ]
    ```
    (The hexadecimal hash following "ethernet_" will probably be different on your system.)
66. **Use set-ipv4-method script to set a static IP address using the connection name identified in step 65**

```
bbb# ./set-ipv4-method ethernet_405fc276b749_cable manual
192.168.1.2 255.255.255.0
```
(note: above is single line with no carriage return.)

67. **Attach the Beaglebone Black to the Ubuntu host using an ethernet crossover cable, reset both**
    Note that the network cards of most modern computers will detect a crossover configuration and automatically switch even with a standard ethernet cable.
68. **Use secure shell to create a connection into the Beaglebone Black**
    ubuntu# ssh root@192.168.1.2
69. **Change to /lib/systemd/system**
    ```
    bbb# cd /lib/systemd/system
    ```
70. **Create file "etherusb.service" with vi editor**
    ```
    bbb# vi etherusb.service
    ```
    Note: if you don't want to use vi, you can use the MMC card reader to creae this file on your Ubuntu pc using gedit.
71. **Press "i" to enter insert mode**
72. **Enter the following into the file:**
    ```
    [Unit]
    Description=Turn on usb0

    [Service]
    Type=oneshot
    ExecStart=/lib/systemd/system/etherusb.sh

    [Install]
    WantedBy=multi-user.target
    ```
73. **Press "ESC" key to enter command mode**
74. **Press ":w" to save the file**
75. **Press ":q" to exit**
76. **Create file "etherusb.sh" with vi editor**
    ```
    bbb# vi etherusb.sh
    ```
77. **Press "i" to enter insert mode**
78. **Enter the following into the file:**
    ```
    #!/bin/sh
    /sbin/modprobe g_ether
    sleep 10
    /sbin/ifconfig usb0 192.168.2.2 netmask 255.255.255.0
    ```
79. **Press ":w" to save the file**
80. **Press ":q" to exit**
81. **Make etherusb.sh executable**
    ```
    bbb# chmod a+x etherusb.sh
    ```
82. **Register etherusb.service with systemd**
    ```
    bbb# systemctl enable etherusb.service
    ```
83. **Test the startup service**
    ```
    bbb# systemctl start etherusb.service
    ```
84. **Verify usb0 is set with ifconfig**
    ```
    bbb# ifconfig
    ```
    You should see an entry for usb0 with address 192.168.2.2
85. **(Optional) Reboot Beaglebone Black and verify usb0 with ifconfig**
86. **Edit the file "/etc/hosts" with vi editor**
    ```
    bbb# vi /etc/hosts
    ```
87. **Press "i" to enter insert mode**
88. **Add the following lines at the end of the file:**
    192.168.1.1 ubuntu.gigether.net
    192.168.1.2 beaglebone.gigether.net

```
192.168.2.1     ubuntu.etherusb.net
192.168.2.2     beaglebone.etherusb.net
```

89. **Press ":w" to save the file**
90. **Press ":q" to exit**