TEXAS INSTRUMENTS

Linux Embedded System Design Workshop

Designing with Texas Instruments ARM and ARM+DSP Systems

Lab Exercises Guide

OMAP3530 Exercises

Workshop Lab Exercises
Revision 3.05
January 2011

Technical Training Organization

Notice

Training Technical Organization

Texas Instruments, Incorporated
6500 Chase Oaks Blvd, MS 8437
Plano, TX 75023
(214) 567-0973

Revision History

October 2006, Version 0.80 (alpha)

December 2006, Versions 0.90/0.91 (alpha2)

January 2007, Version 0.92 (beta)

February 2007, Version 0.95

March 2008, Versions 0.96 (errata)

April 2008, Version 0.98 (chapter rewrites & errata)

September 2008, Version 1.30 (beta 1 & 2)

October 2008, Version 1.30 (beta 3)

February 2010, Version 2.00

August 2010, Version 2.10

October 2010, Version 3.00 / 3.03

December 2010, Version 3.04

January 2011, Version 3.05

Lab Exercises Outline

Lab Exercises

Introduction

3. Configure U-Boot and boot the DVEVM

Application Programming

5. Building programs with GMAKE (and Configuro)
6. *Given*: File ? Audio; *Build*: Audio In ? Audio Out
7. Setup an On-Screen Display (scrolling banner)
   Video In ? Video Out
8. Concurrently run audio and video loop-thru programs

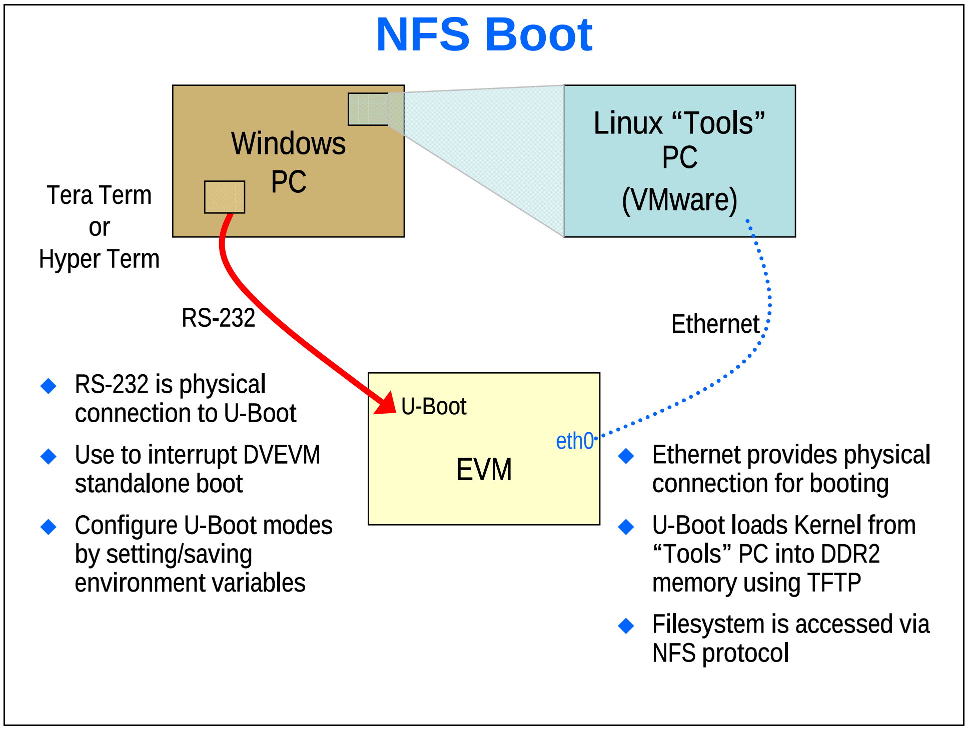
Using the Codec Engine

9. Use a provided Engine (containing local codecs)
10. Build an Engine (given local codecs)
11. Use remote codecs (using a provided DSP Server)
    Swap out video_copy codec for real H.264 codec
12. Build a DSP Server (given DSP-based codecs)

Algorithms

13. Build a DSP algorithm and test it in CCS (in Windows),
    then put your algo into a DSP server and call it from Linux

Lab3 - Experimenting with Linux and U-Boot



Most development for a Linux based target devices, such as the ARM CPU's on the OMAP/Sitara/DaVinci, is done on Linux-based host machines. Developers with Linux PCs can therefore work directly in this environment, but authors using Windows based PCs need either to obtain a new PC running Linux, or employ software that can simulate the Linux environment on top of Windows. In this workshop, VMware is used to create a 'virtual machine' on a windows PC, inside which the Ubuntu operating system can run. In this portion of the lab, the steps to configure Ubuntu on VMware will be implemented. In this lab, the following steps will be taken to set up the software development environment:

Chapter Outline

Lab03a – Start/Configure VMware and Ubuntu Linux

VMware

1. **Launch VMware.**

   On the Windows desktop, **double click** the **VMware** icon.

2. **Open the TTO workshop VMware image.**

   In the VMware Workstation window **Home tab**,

   `Click on the Open Existing VM or Team Icon`

   Open the VMware image file (the name you see might be similar but not exact):

   `C:\vm_images\tto_vm_child_image_(v3.01)\tto_vm_child_v3.01.vmx`

Notes:

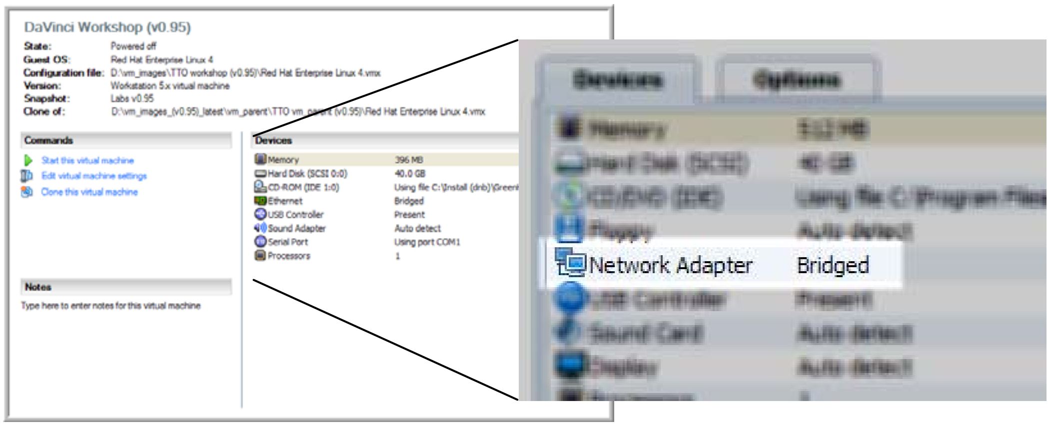
- It's possible your instructor has already started VMware for you. If so, then you may skip this step.
- VM image version v3.01 was current at the time of this writing.
- In USA classrooms, the VMware image is broken into two parts:
  1. *Child* image (~30MB) (C:\vm_images\tto_vm_child_image_(v3.01)\tto_vm_child_v3.01.vmx)
  2. *Parent* image (~20GB) (E:\tto_workshop_v3.00\vm_parent\TTO_vm_parent _(v3.00)

  The child image, specified in this step, depends upon the parent in order to work. Breaking the image into two parts allows us to re-image the C:\ drive being required to reload the entire 20GB for each class.

3. **Verify the Linux networking options are set to 'bridged' mode.**

   This option tells VMware to access the network and obtain its own IP address (other choices involve the Windows PC acting as a router). If not set to 'bridged'

   If you have opened VMware application and the TTO image, you should see the Ethernet setting in the middle of the VMware screen as shown here:



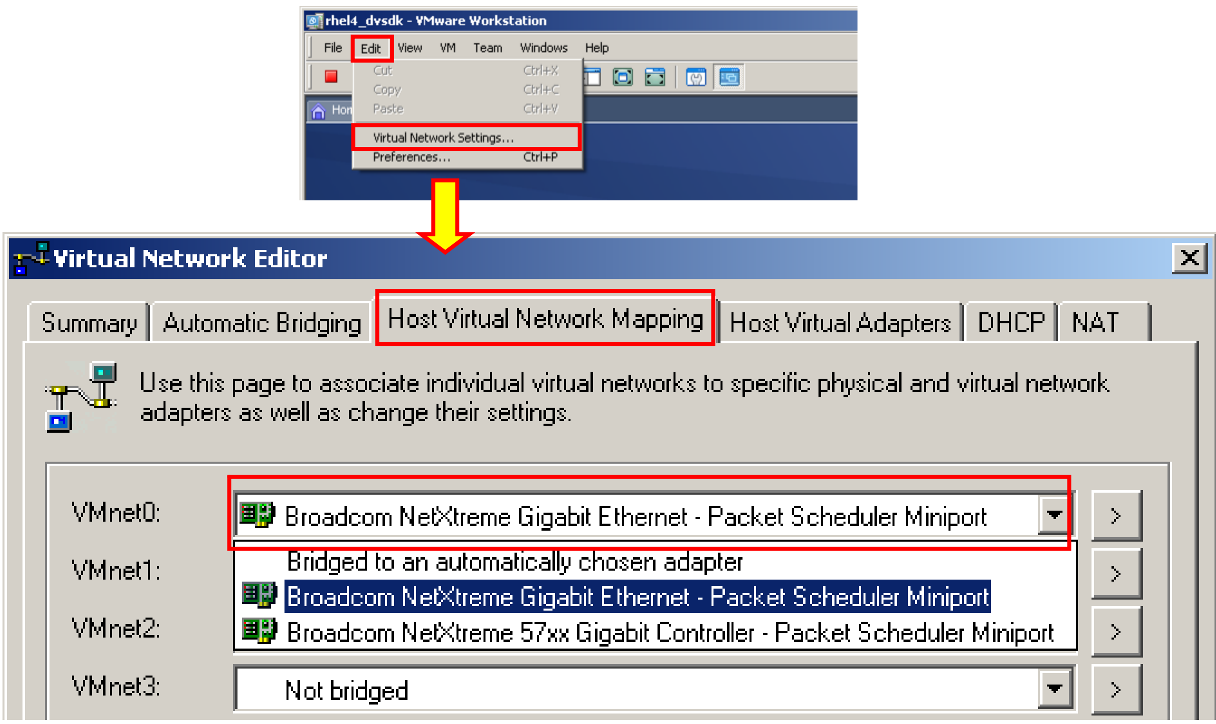
   If you happened to get a little ahead of our instructions and already started the VMware image (which we do in step 6), the easiest way to see this is in the status bar. Just hover over the Ethernet board icon and read the popup message:

Note: If you are using the VMware player, this information is easily found via the top toolbar. In USA classrooms, we use the full version of VMware, though, as opposed to the limited Player version.

4. **Define which of the Ethernet ports on the PC Linux we will use.**

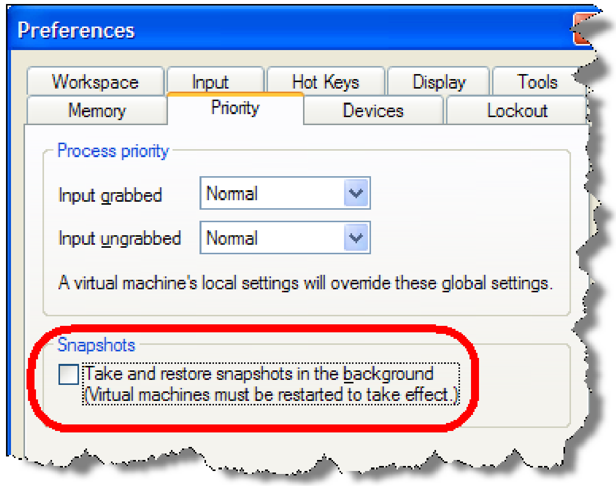
   (Note, this step is required for USA TI classrooms, but may not be needed when using laptops within the USA or for other non-USA locations. Please check with your instructor if you are not sure if this applies to you.)

   From the VMware Workstation menus, select **Edit | Virtual Network Editor…** In the Virtual Network Editor dialog box that appears, go to the **Host Virtual Network Mapping** tab. In the drop box for **VMnet0**, select the **Broadcom NetXtreme Gigabit Ethernet Packet Scheduler Miniport** adaptor, as depicted below:



5. **To improve system speed, disable the VMware snapshot feature.**

   Under **Edit | Preferences**, go to the **Priority** tab, and **uncheck** the **Snapshots** feature. Close the window by clicking on the **OK** button. (If using the VM Player, this option does not apply to you.)



Note:

Your instructor may already have booted your Ubuntu image (in VMware) and left it hibernated (paused). If so, steps 6& 0 might act slightly different.

6. **Start Ubuntu Linux.**

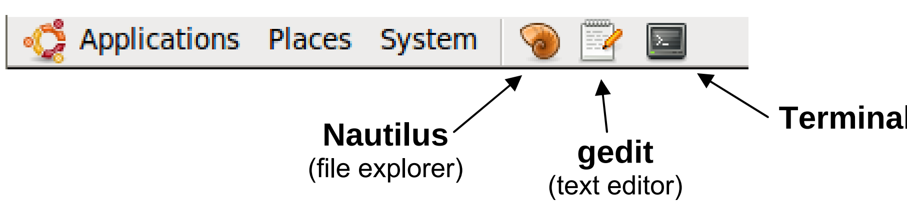
   Click on the green '**Play**' arrow  in the icon bar near the top of the VMware window. (Another way to start the Linux session is to select Start the Virtual Machine in the Commands area). Wait for the boot process to complete (which may take between 2-5 minutes), as indicated by the appearance of the **Log On dialog box**. (If using VM Player, the image is automatically started when opening the VMware Image file.)

   Ubuntu will automatically log you into Linux with a user account. At this point, you will simply see a blank desktop and you can move on to the next step.

   FYI – Ubuntu automagically logged you into the following account – *no login required by you at this time:*

   Ubuntu Userid: **user**
   Password: *none required*

7. **Open a terminal window.**



   The easiest way to open the terminal is to click it's icon on the panel toolbar. You can also find it on the "Applications" menu, but we've placed icons to the three most-used tools onto the toolbar panel.

Lab03b – Install Workshop Lab Files (for your board)

We have installed the appropriate software for your EVM board.

That is, we have worked thru the Getting Started Guides (GSG) for each of the boards (OMAP3530 and AM3517) into the same VMware image, because they both use the same DVSDK/SDK (software development kit) and version of community Linux.

Since the DM6446 uses a different DVSDK, we chose to install its software libraries (and MontaVista Linux) into a separate VMware image.

In this part, you will install the workshop labs/solutions files per the board you have chosen to work on during class. Additionally, we will configure/verify a couple of environment settings.

Installing Workshop *Labs* and *Solutions* Files

8. Verify the `shared` folder is enabled.

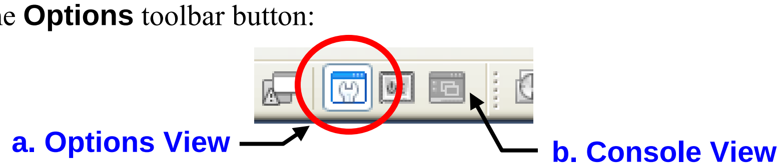
Let's try simply listing the files in the shared folder. If there aren't any files, we may need to enable this VMware feature.

```
ls –l  /mnt/hgfs/shared
```

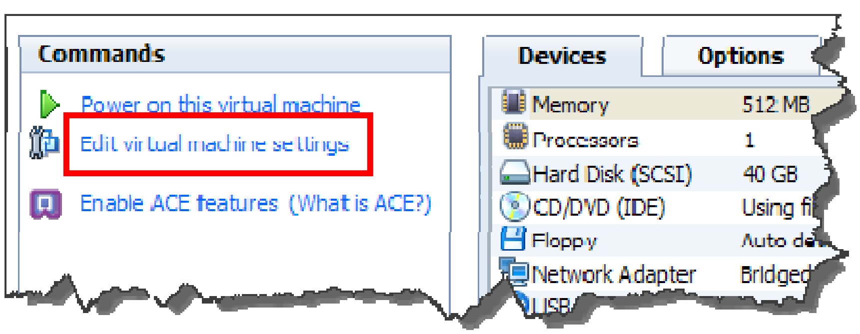
If this doesn't work, shared folders are not enabled. Continue with the next step to enable shared folders.

9. If needed, enable shared folders.

If VMware Workstation is running (and it probably is, at this point), go to "options" view by clicking on the **Options** toolbar button:



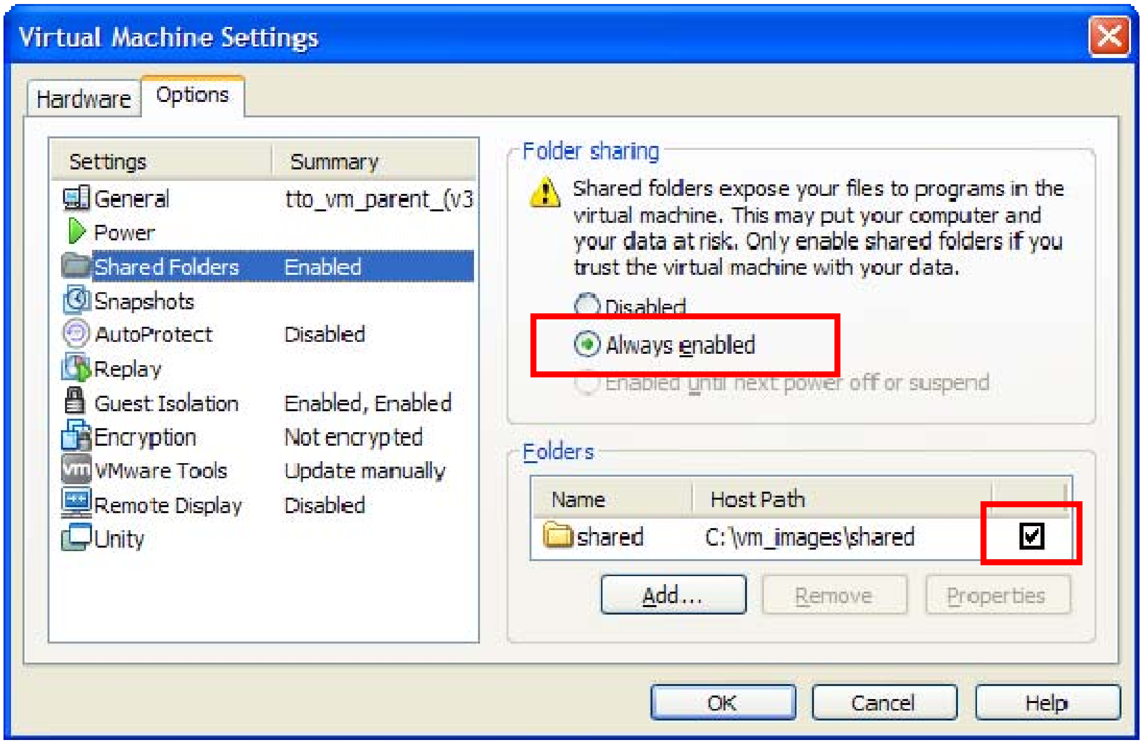
Click on **Edit Virtual machine settings:**



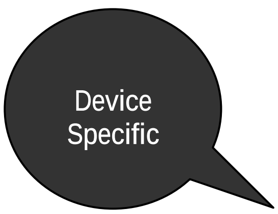
And then …

When finished enabling shared folders, simply click the "Console View" button in VMware to get back to the command line.

Make sure that **Shared Folders** are *Always enabled*:



10. Copy lab files from Windows/VMware shared directory.



To keep things simple, for the OMAP3530 and AM3517 VMware image, everything but the lab files have been installed. Rather than putting lab files for both target boards in the user folder, we have provided you two tar files.

```
cd /home/user
cp /mnt/hgfs/shared/TTO_Linux_SOC_Workshop_labs_omap35_v3.xx.tar.gz .
```

Options:

- For the AM3517 choose: `TTO_Linux_SOC_Workshop_labs_am3517_v3.xx.tar.gz`
- Rather than seeing a file with `v3.xx`, choose the latest revision available; e.g., `v3.03`.
- DM6446 users can ignore this step.

11. Untar the lab files into the `/home/user` folder.

In the steps below, make sure you use the file you copied in the previous step.

```
cd /home/user
tar –xzf TTO_Linux_SOC_Workshop_labs_omap35_v3.xx.tar.gz
```

After unzipping, you should have two new folders in your `/home/user` folder. If not, please consult with your instructor.

```
/home/user/labs
/home/user/solutions
```

Device Specific

12. **Verify you have installed the correct files for your EVM platform.**

    (You can skip this step if you are following the DM6446 labs.)

    Check that the readme file exists in your new labs (and/or solutions) folder. We use the readme file to confirm the platform supported – along with the workshop labs/solutions version number.

    `/home/user/labs`

    `Readme_omap35_labs_v3.xx.txt`

    or **`Readme_am3517_labs_v3.xx.txt`**

Device Specific

13. **Add symbolic link to `targetfs` directory.** *(You can skip this step if you are doing the DM6446 labs.)*

    Finally, we need to add a Linux symbolic link for our **`targetfs`** directory.

    ```
    ln -s /home/user/psp_rebuild_omap3/linux_filesys /home/user/targetfs
    ```

    or

    ```
    ln -s /home/user/psp_rebuild_am3517/linux_filesys /home/user/targetfs
    ```

    This Linux command (small LN) creates a symbolic link, similar in some ways to a Windows shortcut. With this link, we can now refer to the `/home/user/targetfs` directory in our workshop instructions and the correct folders/files will be referenced on each of your systems, no matter which EVM you are using.

    This is also the directory we are “exporting” (i.e. network sharing). We already set this up for you in Linux by editing the /etc/exports file. This was required because since this is the folder used as the *nfspath* – that is, we will use this folder (via the network) as the *root filesystem* for our EVM.

Installing kernel modules to the *targetfs*

14. **Install the kernel modules and `loadmodules.sh` script to the target filesystem.**

    We have conveniently placed the kernel modules and scripts into the lab00 folder. All you need to do is run the install script located in that folder.

    ```
    cd /home/user/labs/lab00_install_scripts
    ./install.sh
    ```

    This will copy the files contained in this folder over to our workshop directory in the target filesystem (`/home/user/targetfs/opt/workshop`). Later on we’ll discuss what these files are used for; for now, we just want to copy them into place so they’ll be there when we need them.

Lab03c – Image SD/MMC card (to boot EVM)

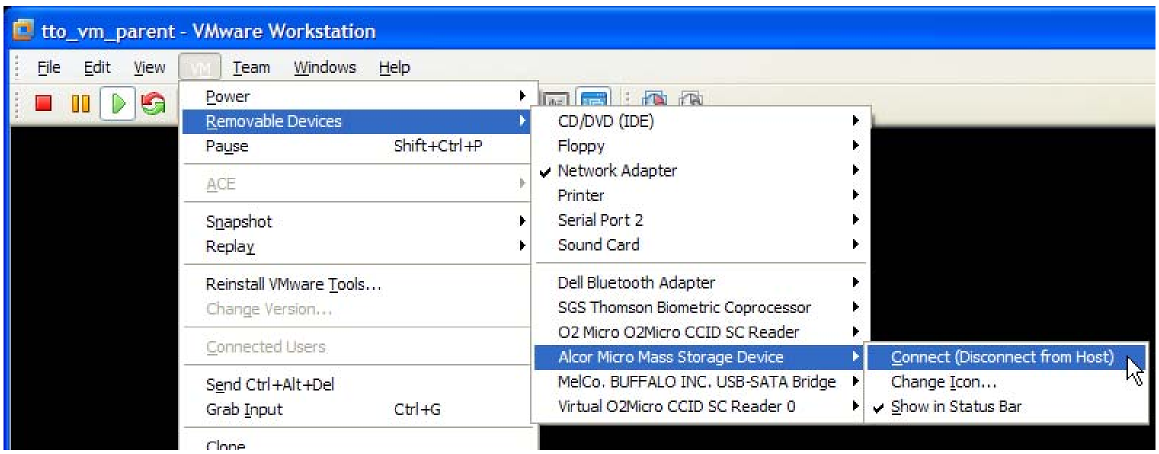
15. **Plug USB Flash SD/MMC Card reader into a USB port on your computer.**

    You may see a dialogue box talking about "Removable Devices" – just click OK and continue.

16. **Connect the SD/MMC flash card reader to the Ubuntu virtual machine.**

    If USB Flash Card reader is mapped to Windows host, select:

    VM → Removable Devices → ***<Your Flash Reader>*** → Connect…



Note: Your SD/MMC card reader may show up as a slightly different name, depending upon the brand of reader you are using

17. **Open a terminal window in Ubuntu** (if one is not already open)**.**

18. **Move to the Lab03a directory.**

```
cd ~/labs/lab03_build_sd
```

19. Determine SCSI device node for USB SD card reader

```
(user@ubuntu) #  sudo sg_map –i

When prompted for sudo password, (press enter)
```

You should see a table similar to the following:

```
/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     1.0
```

Depending on the SD/MMC Reader used, it may appear differently, but will likely be the last device on the list.

Write down the Linux device node (i.e. virtual file name) for the card reader:

Your device node: ________________________________ (most likely, `/dev/sdb`)

20. Insert a 2GB SD/MMC card into the USB Flash reader (if not already done)**.**

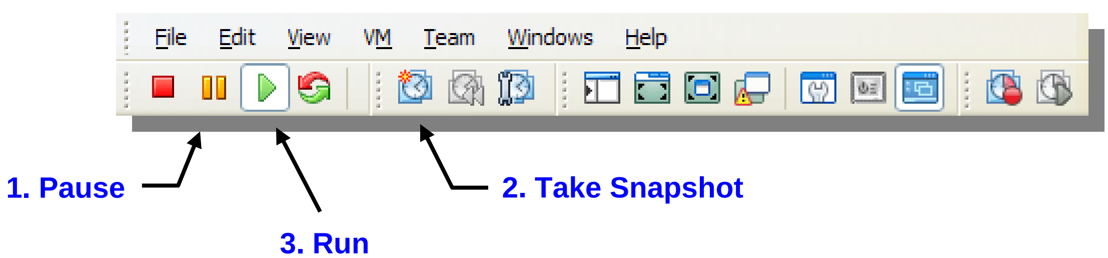
Caution

> Read the following step and comments very carefully, specifying the wrong `/dev/sdx` device node could cause permanent damage to your system!

21. Take a VMware snapshot. (*Only full version of VMware Workstation supports snapshots.*)

Because this step could erase the wrong drive in your system, let's make a snapshot copy of our virtual hard drive. This can be done many ways, but we suggest this simple 3-step procedure – which uses three different VM toolbar buttons:

1. Pause your Linux VMware PC.
2. Take a snapshot.
3. Un-pause (that is, Run) your Linux VM, again.



22. Execute the `build_sd.sh` script.

Run the build script <u>using the device node from step 19</u> (page 3-10). If prompted for a sudo password, simply press enter (blank password).

```
(user@ubuntu):  SCSI_DEV=/dev/sdb ./build_boot_sd.sh
```

When asked to "confirm", press "y" and [ENTER].

It should take less than a minute for the script to complete. The script automates these steps:

– Un-mounts partitions (if any) that Ubuntu automatically mounts to the desktop

– Reformats and formats the SD/MMC card for two partitions
(though we'll only use one, for now)

– Temporarily mounts new partitions and copies three files onto the 1st partition:

```
MLO          (X-loader – 2nd level bootloader)
u-boot.bin   (uboot – 3rd level Linux bootloader)
uImage       (Linux kernel)
```

In the next part of the lab, we'll use the MMC card to boot the EVM.

Lab03d – Talking to the EVM

23. Start TeraTerm.

On the Windows desktop, **double click** on the **TeraTerm** icon.
The TeraTerm serial configuration file dvevm.ini, in the TeraTerm program folder has already been set up with the following necessary configuration states:

```
Bits per Second: 115200
Data Bits: 8
Parity: None
Stop Bits: 1
Flow Control: None
```

24. Insert the SD/MMC card into the EVM.

If you haven't already done so, remove the SD/MMC card you formatted in step 22 from the card reader.

```
Insert the card in the EVM's SD/MMC card slot
```

The card should go into the slot "label up" – SD card "pins" down. On new boards, the slot is tight, so you need to make sure and line it up very straight as you slide the card into it.

25. Connect RS-232 serial cable.

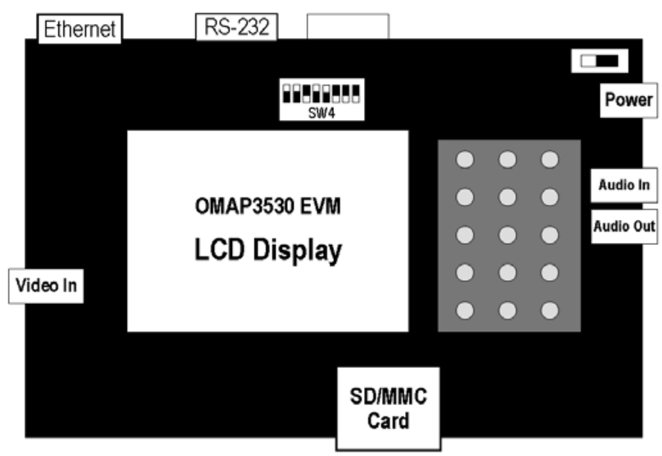
If not already done, please connect the serial cable. (If unsure how to do this, please as your instructor (or refer the EVM Quick Start Guide).

```
Connect RS-232 cable between the EVM and PC RS-232 port

Note: For OMAP3530 EVM, please use the UART1/2 connector.
```

Device Specific

26. Verify EVM Hardware Configuration

- Is the EVM powered off?
- Verify the switch settings (for proper booting) of the EVM - where does board find MLO and uboot.bin?



OMAP3530 EVM switch S4

```
SD/MMC card:     0010 0111
On-board NAND:   xxxx xxxx
```

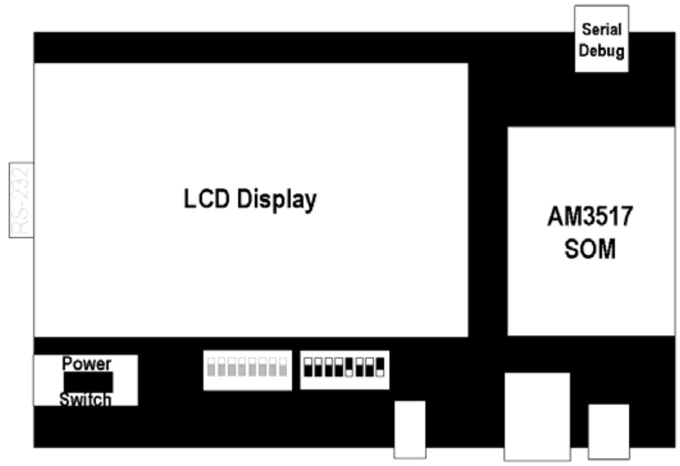
(AM3517 switch settings continued on next page.)

AM3517 EVM switch S7

```
SD/MMC card:     0000 1001
```

Setting the first and fourth switches on, while the others are off, tells the board to boot using the MMC card.

```
sw 7-1: on
sw 7-2: off
sw 7-3: off
sw 7-4: on
sw 7-5: off
sw 7-6: off
sw 7-7: off
sw 7-8: off
```



These switches modify the boot mode pins on the AM3517, which are used by the ROM bootloader (1st stage) to use the XLOADER (2nd stage bootloader) found on the first partition of the MMC card. If all switches are off, the device will boot using the XLOADER found in the EVM's onboard NAND flash.

To learn more about the switches (and configuration) of the AM3517 board, visit:

http://processors.wiki.ti.com/index.php/GSG:_AM35x_EVM_Hardware_Setup

27. **Start the EVM – by plugging in the Power cable.** (You may also need to toggle switch next to power cord.)

    Power on the EVM board and press any key to interrupt U-Boot's boot sequence.

    ```
    Press any key    (to stop Linux from booting)
    ```

    At this point, the EVM U-Boot terminal prompt (`DaVinci EVM#, OMAP3#, AM3517#`) should be visible in the TeraTerm session window.

In a few minutes we'll setup U-boot and get the board running …

Lab03e – Verify Networking and Record IP Addresses

Connecting to the Network

28. Make sure the Ethernet cable is connected between your EVM and the PC where you're running VMware.

If you're direct connecting the VMware image to the target EVM, then make sure the SD/MMC card you just programmed is inserted into the EVM and then power-on the EVM board (we don't care what it does at this point – that will be handled in the next section).

On the other hand, if you're using a switch or router, simply make sure that the swtch is up-and-running and connected to the EVM and PC.

29. Record the Windows Ethernet address.

This information will be used to test the Linux Ethernet connection in the next step. In the Windows **system tray** (right side of the Windows task bar) **double click** on the Local Area Connection 2 icon: 

From the **Support tab** of the dialog box that popped up, write the noted values below. Close the window when done recording this value.

IP Address ________________________

Note, If there are two wired LAN icons in the Windows taskbar, you should choose the one with the IP address: 192.168.1.39

30. Determine the Ubuntu Ethernet address.

You must have the Ethernet cable plugged in to the EVM and the board powered on or you will get an error during this step. (You should have connected the Ethernet cable in step 28.)

In the terminal window, run **ifconfig** by typing: **/sbin/ifconfig** ↵ and transcribe the IP address below. (Alternatively, we've set our $PATH statement so that you can just type *ifconfig*.)

IP Address _____________________ (Note, it will be called "inet" in the Linux response)

31. Test the Linux Ethernet port:

Ping the Windows Ethernet port to verify that both are working. In the terminal, type:

```
ping <IP_Address>
```

Where *IP_address* is the value recorded in step 29 above. The response should look like:

```
[user@localhost ~]$ ping 192.168.1.39
PING 192.168.1.39 (192.168.1.39) 56(84) bytes of data.
64 bytes from 192.168.1.39: icmp_seq=0 ttl=128 time=2.00 ms
64 bytes from 192.168.1.39: icmp_seq=1 ttl=128 time=0.675 ms
64 bytes from 192.168.1.39: icmp_seq=2 ttl=128 time=0.800 ms
```

In Linux, you need to halt the ping command using:

```
<Ctrl> C to halt the pinging
```

These are the IP addresses we plan to use in this workshop:

Windows PC:	192.168.1.39
Ubuntu Linux:	192.168.1.1
EVM target:	~~192.168.1.41~~ dynamically set

Lab03f – Configure U-Boot and Boot the EVM

32. Return to Windows and TeraTerm.

Since VMware implements a complete virtual PC when the cursor is within its borders, it is necessary to move the cursor outside the VMware frame so that the use of **Alt + Tab** will invoke the underlying Windows OS and allow control to pass from the VMware application to another Windows program. Then, hold down the Alt key and repeatedly pressing Tab until the **TeraTerm** application is selected.
Release the Alt key to complete the selection.

Based on where we left things earlier in the lab, you should be at the U-Boot prompt. If this is not the case, power-cycle the board and then stop U-Boot from booting into Linux by hitting any key.

Device Specific

33. Run the TeraTerm macro to setup the EVM's U-Boot mode.

From TeraTerm, select **Control | Macro**. From directory **C:\Program Files\TTERMPRO** select the file associated to your board:

```
DM6446 DVEVM:    tto_uboot_setup.ttl
OMAP3530 EVM:    tto_uboot_setup_3530.ttl
AM3517 EVM:      tto_uboot_setup_3517.ttl
```

If the macro pauses, simply hit [ENTER] in the terminal window to continue with the questions below:

As the macro runs, make the following selections:

- Use Default NFS Server IP Address: 192.168.1.1 **Yes**
- Boot Static or Dynamic? [Yes= Dynamic (dhcp), No=Static)] **Yes**
- Root Filesystem from NFS or MMC? (Yes=NFS, No=MMC) **Yes**
- Use default NFS path? (/home/user/targetfs) **Yes**
- Kernel from TFTP or Flash/MMC? (Yes=TFTP, No=Flash/MMC) **Yes**
- For TFTP boot, use the default Kernel image filename? **Yes**
- Save bootargs? **Yes**
- Boot Linux now? (No, we'll do this manually) **No**

34. Test network connection from EVM to Ubuntu VMware image.

This is a good thing to check, since we plan to boot the EVM across the network – that is, we plan to get the root filesystem (and maybe the Linux Kernel) for the EVM from our Ubuntu Linux VM image.

Run the **ping** command from Uboot.:

```
ping 192.168.1.1
```

It should respond with: `Connection is alive`

35. Examine the EVM's Linux environment.

The **printenv** ↵. reports the current state of the U-Boot variables. You should be able to see the changes we made with our interactive TeraTerm script.

36. Save the new U-Boot settings.

Changes to the environment must be saved to the Flash to remain active after power-cycling the EVM hardware. This is done automatically by the macro when you answer **Yes** to the 'save bootargs' question.

To manually preserve the bootargs, type:

save ↵

37. Take Home exercise…

Review the macro by opening the file in any text editor. While not commented in detail its code should be easy to understand if one knows the U-Boot options in general.

38. Boot / Reboot the EVM.

Power-cycle the DVEVM or type boot ↵ to restart the EVM with the new environment settings. When boot completes (you can watch it in Tera Term – should take a few minutes), **log in as root user**; no password is needed. *Note: if during bootup "kernel panic" is reported, ask the instructor for assistance.*

Your Windows terminal (i.e. Tera Term) is now connected to the "target" Linux running on the EVM's ARM processor.

Sidebar

It is common practice to log into a host Linux PC as a user (i.e. not as the root user). Conversely, it is also common practice to log into a development board, like the EVM using the root user. In embedded applications, there often only exists a single user (root).

39. Verify shared file system between Ubuntu and EVM.

Since any file change to the root directory of our EVM board will be reflected in Ubuntu Linux, let's give it a try by creating a new file (or updating its timestamp) using the Linux "touch" command.

From Tera Term (which is now logged into the EVM board):

```
root@omap3evm:~# cd  /                                    moves you to root
root@omap3evm:~# touch putfileatroot.txt create a new empty file at root
```

Now, let's look for this file on the NFS source directory; that is, in the target filesystem on our Ubuntu PC. To do this, list the files of the target filesystem **from the Ubuntu terminal session** (note: be careful to be in the correct window, as there are two that can be easily mistaken for each other) you started earlier:

```
[user@localhost user]# cd /home/user/targetfs
[user@localhost user]# ls -la
```

You should see the *putfileatroot.txt* in your listing, with the current date and time stamp (you could always try the Linux *date* command if you'd like to change it to your time zone). Note, you can see the same directory (and file) from both environments. Similarly, when we create new app's within Ubuntu Linux, if they are created (or copied to) our target filesystem folder, they're immediately available at our NFS mounted EVM target.

Review

To summarize, the root path of the EVM is set to a path inside the User's home directory. Fill in the box below indicating the path within Ubuntu Linux where the EVM board's root path is associated.

EVM Board "Target"		Ubuntu Linux "VMware Image"
/	=	

How did this association get made? ____

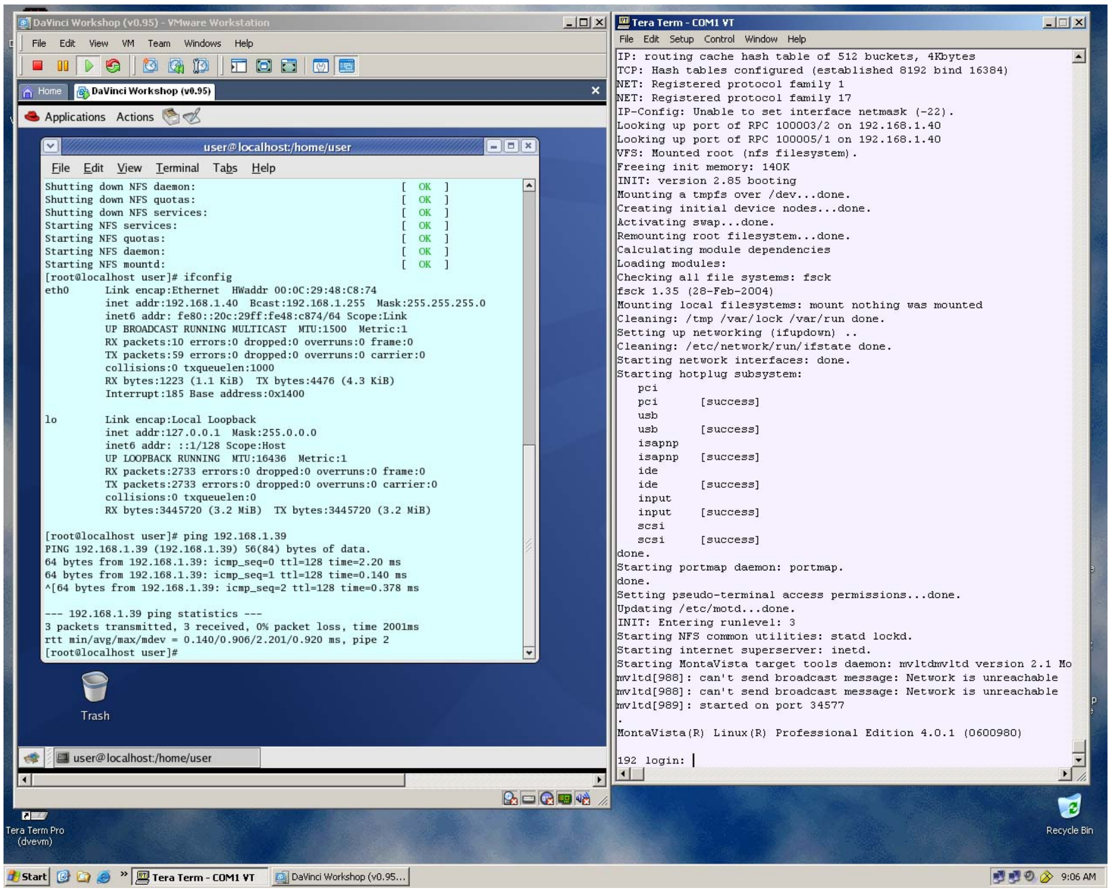
____

What is the advantage using an NFS (networked) mounted filesystem versus using the hard drive (or flash drive) built into the DaVinci board? _____________________________

____

(Optional) Lab03g – Try Other Boot & VM Options

a. Arrange your Desktop windows to show both terminals side-by-side. This might make it easier to keep from confusing one terminal versus another. *(Doesn't really work well on small laptop computer screens, but works great with larger monitors.)*



Device Specific

b. Try booting up the board with other combinations of options:

- If you have a router, you could try dynamic IP addresses (vs Static IP).
- On DM6446 DVEVM, try … **Kernel/Root filesystem: tftp/nfs, tftp/hdd, flash/hdd, flash/nfs.** (Note, though, you will need to update the flash on the board, first.)
- On the OMAP35 or AM35 EVM's:

	MLO (i.e. xloader)	u-boot	Kernel	Filesystem	Comments
✓ 1	mmc	mmc	tftp	nfs	This was lab Lab03e.
2	mmc	mmc	mmc	nfs	Re-run TeraTerm setup macro to choose these options.
3	flash	flash	tftp	nfs	Need to program the flash first, see Software Dev'l Guide instructions. Then, re-run the TeraTerm setup macro.
4	flash	flash	flash	nfs	
5	mmc	mmc	mmc	mmc	Later we'll examine copying the filesystem to SD/MMC.

Lab 5 - Building Programs with gMake

Goal

Welcome to the compulsory "Hello World!" lab exercise. Here we will begin our exploration of Sitara\DaVinci\OMAP software programming tools. In this lab, you will:

- Create a simple X86 makefile for building a specific program ("Hello World").
- Write the makefile code to consume a "package" delivered by TI (Configuro)
- Explore and analyze a more complex, generic makefile that will be used throughout the rest of this workshop.
- Execute the "Hello world!" application on both the x86-based Linux host system and the ARM-based target system using Linux terminals.

Outline

Lab 5 – Makefiles and Configuro

- Part A – Building a Simple Makefile
- Part B – Using Built-in and User-defined Variables
- Part C – Using Configuro to Consume a Package
- Part D – Using the Final DaVinci Workshop Makefiles

Optional

- Part E – Analyzing the Final DaVinci Workshop Makefiles

- Time: 60-75 minutes

Lab05ab_basic_make

Big Picture

In part A of the lab, you will build your first basic makefile – basically turning command line execution into gMake rules. In Part B, you will increase the usability of your makefile by adding built-in variables and user-defined variables. This will provide you with a fundamental understanding of how makefiles work.

Procedure

Lab Prep – Examine the directory contents and app.c

1. **Open a terminal in the Linux Host Computer.**

   Log into the Linux Host (i.e. desktop) Computer. Open a terminal window clicking on the "Terminal" toolbar icon.

   You will begin in the `/home/user` directory (the home directory of the user named "user"), also represented by the tilde (~) symbol.

2. **Locate the `labs` directory and list its contents.**

   Descend into to the `/home/user/labs` directory using the "cd" command. ("cd" is short for "change directory").

   Use the "`ls`" (lower case "LS") command to ***list*** the contents of this directory:

   ```
   ls
   ```

   At any time, if you're curious about which directory you are in, use the Linux "`pwd`" command. This stands for "path working directory":

   ```
   pwd
   ```

   The `labs` directory is the working directory for the lab exercises and contains all the starter files needed for this workshop. (Note, solutions for each lab can be found at `/home/user/solutions`).

   In addition to all of the lab folders, one of the additional files at this level is named `setpaths.mak`, which you will use later in this lab. `setpaths.mak` contains absolute paths for the locations of various tools and packages that will be used to build projects throughout the workshop. More on this later.

   For this workshop, the proper file paths have already been configured for you. However, when you take your labs and solutions home to work on them further, you may need to modify `setpaths.mak` in order to build correctly on your system. (Note, the DVSDK uses the file named *Rules.make* for the same purpose as our *setpaths.mak*.)

3. Examine the contents of the `lab05abc_basic_make` directory.

```
cd ~/labs/lab05ab_basic_make
```

- or -

```
cd /home/user/labs/lab05ab_basic_make
```

- or, since you're probably already in the labs directory -

```
cd lab05ab_basic_make
```

List the contents of this directory. The `lab05ab_basic_make` folder contains only one directory, `/app`. (Later, as our lab exercises become more complex, some projects will have multiple directories at this level.)

4. Examine `app.c` in the `lab05ab_basic_make/app` directory.

Descend into the `app` directory. Examine the C source file `app.c` which prints the string "Hello World" to standard output.

```
cd app
gedit app.c
```

Part A – Using the Command Line and Creating a Simple Makefile

In this part, we will simply use the GNU compiler (`gcc`) from the command line to build the "Hello World" example and run it. Then, we'll place these commands into a basic makefile and run the makefile. In the next part, we'll use built-in and user-defined variables.

5. Build and run "Hello World" from the command line.

Make sure you are in `lab05ab_basic_make/app` folder.

To compile `app.c`, type the following command:

```
gcc –g –c app.c –o app.o
```

gcc	= GNU C compiler (command)
–g	= symbolic debug (compiler option)
–c	= (fill in answer below)
app.c	= file to compile (kind of "dependency" or "prerequisite")
-o	= output filename is next (compiler option)
app.o	= output file (the "target")

In the above gcc command, can you name the target, dependency and command?

- **Target =** ________________
- **Dependency =** _________________
- **Command =** ___________________

6. Use the "`man`" command to look up `gcc`.

To find the parameters for any standard C functions or Linux commands, you can use the "`man`" (short for "manual") command. Let's try it on gcc:

```
man gcc
```

What does the –c option (from step 5) tell the compiler to do?

____

To quit the *man* page, type "q" at least once (depending on where you are in the page, you might need to type "q" multiple times).

7. Link the object file and produce the final executable.

Next, link the object file (`app.o`) to create the executable `app.x86U`:

```
gcc –g app.o –o app.x86U
```

Now run the executable:

```
./app.x86U
```

You should see "Hello World" displayed in the command window.

The extension used for the output file (`.x86U`) indicates we are building for the x86 (or host PC). In the future, we will build for the ARM target on the EVM and it will have a different extension (more on this later).

Note: For those of you who know Linux well, you can skip this explanation. For the rest …

> `./` before the name of an executable tells Linux to look for the program in the current directory.

We use this as it is the proper way to specify the path of the file to be run. Just in case you make a mistake and forget to include the `./`, we added it to our Linux $PATH environment variable, so Linux will still be able to find your program.

---

8. "Clean" the existing executable (.x86U) and intermediate (.o) files.

Type the following to remove the files generated by the gcc commands you executed:

```
rm –rf app.x86U
```

```
rm –rf app.o
```

This removal of files mirrors what a "clean" macro or rule might do. We'll actually add a rule shortly to accomplish this in our `makefile`.

9. Examine "starter" `makefile`.

The current `makefile` in the `lab05ab_basic_make/app` directory simply contains comments and placeholders for the code you will write. Using your favorite editor, open the `makefile`. For example:

```
gedit makefile
```

10. Create rules for `app.x86U` and `app.o` in your `makefile`.

Remember, a rule is made up of a target, dependency(ies) and command(s). For example:

```
target : dependency
        CMD
```

Also note that the commands are tabbed over (at least one tab).

Create the rule for `app.o` in the area of the `makefile` with the header comments specifying the intermediate (.o) rule (as shown below). We'll help you with the rule for `app.o`, but `app.x86U` is up to you. If you get stuck, look back at the chapter material, ask the instructor for help or peek at the solution.

For `app.o`, type in the following rule. We will use the absolute path of `gcc` for now and later turn it into a variable:

```
# -----------------------------------------------------
# ------ intermeditate object files rule (.o) -------
# -----------------------------------------------------
app.o : app.c
     /usr/bin/gcc –g –c app.c –o app.o
```

app.o	= target
app.c	= dependency
/usr/bin/gcc -g …	= command

11. Type in the rule for .x.

Next, type in the rule for `app.x86U` ABOVE the rule for `app.o` in the area specified for the (`.x`) rule. Make sure you use the –g compiler option in the `.x` rule.

12. Test your makefile.

Close makefile and type the following:

```
make
```

After running make, list the current directory.

```
ls
```

Do you see a new `app.x86U` executable? Run it:

```
./app.x86U
```

Do you see "Hello World"? If so, your rules work. Next, let's add a few more rules…

13. Open `makefile` in a different Linux process.

Stop. Before you open `makefile` again, try opening it in a different Linux process by typing in the following:

```
gedit makefile &
```

The "&" tells Linux to open the makefile in a separate process (window). When you edit a file, you can simply click Save, then click inside the terminal window and run it without having to re-open the makefile. Handy – and could save you some time.

14. Create a "clean" rule in your makefile

Whenever you run *gMake*, it will search and note the timestamps of the source files and executables and won't run if everything is up to date. So, it is common to create a "clean" rule that removes the intermediate and executable files prior to the next build.

In the `makefile` (underneath the comment header for "clean all"), add the following `.PHONY` rule for "clean" (these are the same commands you used earlier on the command line):

```
.PHONY : clean
clean  :
        rm –rf ___.x86U
        rm –rf _______
```

`.PHONY` tells *gMake* to NOT search for a file named "clean" because this is a phony target (i.e. it is not a file that needs to be searched for or created). In a large and complex makefile, this actually saves some compile time (plus, it is just good practice to use `.PHONY` when the target is not an actual file). The two files are the final executable and the intermediate object file.

15. Create an "all" rule in your `makefile`.

When *gMake* runs without any rules specified (i.e. you just type "`make`" on the command line), it will make (by default) the first rule in the `makefile`. Therefore, it is common to create an "all" rule that is placed first in the `makefile`. Our example only has one final target (`app.x86U`), so "all" doesn't make as much sense now. However, when we move to the `makefile` for the ARM target on the EVM, we'll have multiple targets to build and it will be more useful.

In the `makefile` (under the comment header for "make all"), add the following .PHONY rule for "all":

```
.PHONY : all
all : app.x86U
```

Close `makefile` and let's run it…

16. Run gMake to create the executable `app.x86U`.

On the command line, type in the following:

```
make
```

gMake will probably tell you that the files are "up to date" and there is nothing to do. So, you must run "clean" before you build again. Type:

```
make clean
```

and then:

```
make
```

or:

```
make all
```

gMake runs the first rule in the makefile which is the "all" rule. This should successfully build the `app.x86U` executable.

Note: *gMake* assumes the name of the make file is `makefile` or `Makefile`. *gMake* also looks for the FIRST makefile it finds. So, to be safe, you might want to capitalize Makefile because capital "M" comes before lower-case "m" alphabetically. You can also use a different name for the makefile – e.g. `my_makefile.mak`. In this case, you need to use the following command to "force" the use of a different make file name:

```
make –f my_makefile.mak
```

---

17. Run `app.x86U`.

You should see "Hello World" again. Ok, now that we have the simple makefile done, let's turn it up a few notches…

18. Review the different ways to run *gMake*.

As a review, you can run *gMake* in several ways:

`make`	(makes the first rule in the make file named makefile or Makefile)
`make <rule>`	(makes the rule specified with <rule>, e.g. "make clean")
`make –f my_makefile`	(forces the use of a make file named my_makefile)

Part B – Using Built-in and User-Defined Variables

In this part, we will add some user-defined variables and built-in variables to simplify and help the makefile more readable. You will also have a chance to build a "test' rule to help debug your makefile.

19. Add CC (user-defined variable) to your makefile.

Right now, our x86 `makefile` is "hard coded". Over the next few steps, we'll attempt to make it more generic. Variables make your code more readable and maintainable over time. With a large, complex makefile, you will only want to change variables in one spot vs. changing them everywhere in the code.

Add the following variable in the section of your `makefile` labeled "user-defined vars":

```
CC := $(LINUX86_GCC)
```

`CC` specifies the path and name of the compiler being used. Notice that `CC` is based on another variable named `LINUX86_GCC`. Where does this name come from? It comes from an include file named `path.mak`.

Open `path.mak` and view its contents. Notice the use of `LINUX86_GCC` variable and what it is set to.

Whenever you use a variable (like `CC`) in a rule, you must place it inside `$( )` for *gMake* to recognize it – for example, `$(CC)`.

After adding this variable, use it in the two rules (`.x` and `.o`). For example, the command for the `.x` rule changes from:

```
gcc –g app.o –o app.x86U
```

- to this -

```
$(CC) -g app.o –o app.x86U
```

20. Apply this same concept to the `.o` rule.

21. Add include for path.mak.

In the "include" area of the makefile, add the following statement:

```
-include ./path.mak
```

22. Test your makefile: clean, make and then run the executable.

23. **Add `CFLAGS` and `LINKER_FLAGS` variables to your makefile.**

    Add the following variables in the section of your `makefile` labeled "user-defined vars":

    ```
    CFLAGS := -g
    LINKER_FLAGS := -lstdc++
    ```

    `CFLAGS` specifies the compiler options – in this case, -g (symbolic debug).
    `LINKER_FLAGS` will tell the linker to include this standard library during build.
    (The example option `–lstd++` specifies the linker should include the standard C++ libraries.)

    Use these new variables in the .x and .o rules in your makefile.

24. **Test your makefile.**

25. **Add built-in variables to your .o rule.**

    As discussed in the chapter, *gMake* contains some built in variables for targets (`$@`), dependencies (`$^` or `$<`) and wildcards (`%`). Modify the `.o` rule to use these built-in variables.

    The `.o` rule changes from:

    ```
    app.o : app.c
            $(CC) $(CFLAGS) –c app.c –o app.o
    ```

    – to –

    ```
    %.o : %.c
            $(CC) $(CFLAGS) –c _____ -o _____
    ```

    Because we only have ONE dependency, ***use the $< to indicate the first dependency only***. Later on, if we add more dependencies, we might have to change this built-in symbol. `%` is a special type of *gMake* substitution for targets and dependencies. The `%.o` rule will not run unless a "`filename.o`" is a dependency to another rule (and, in our case, `app.o` is a dependency to the `.x` rule – so it works).

26. **Add built-in variables to your .x rule.**

    The .x rule changes from:

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

    - to -

    ```
    app.x86U : app.o
            $(CC) $(CFLAGS) $(LINKER_FLAGS) _____ -o _____
    ```

27. **Don't forget to add the add'l `LINKER_FLAGS` to the .x rule.**

28. **Test makefile.**

29. Add a comment to your .x rule.

Comments can be printed to standard I/O by using the `echo` command. In the `.x` rule, add a second command line as follows:

```
@echo; echo $@ successfully created; echo
```

The `@echo` command tells *gMake* to echo "nothing" and don't echo the word "echo". So, effectively, this is a line return (just like the echo at the end of the line). Because built-in variables are valid for the entire rule, we can use the `$@` to indicate the target name.

Test `makefile` and observe the echo commands. Did they work? As usual, you might need to run "make clean" before "make" so that *gMake* builds the executable.

30. Add "test" rule to help debug your makefile.

Near the bottom of `makefile`, you'll see a commented area named "basic debug for makefile". Add the following `.PHONY` rule beneath the comments:

```
.PHONY : test
test:
        @echo CC = $(CC)
```

This will echo the path and name of the compiler used. Try it. Does it work?

You can also add other echo statements for `CFLAGS` and `LINUX86_GCC`. This is a handy method to debug your makefile.

Close your makefile when finished.

Lab05c_x86_configuro

Part C – Using Configuro ....

In this part, we will use the Configuro tool to consume a package delivered by TI. This package will allow us to use the System_printf( ) command found in app.c. Because content is delivered by TI and 3rd parties as "packages", it is important to understand the basics of using Configuro.

31. **Copy `makefile` from your previous lab directory to the new lab directory.**

    From the `lab05ab_basic_make/app` directory, type:

    ```
    cp makefile ../../lab05c_x86_configuro/app
    ```

    This should copy your makefile to the next lab's directory.

32. **Change directories to `/labs/lab05c_x86_configuro/app` directory.**

    This is the working directory for Part C of the lab. Do a listing of this directory. You'll see the following files:

    - **`app.c`** – updated to use System_printf( )
    - **`app_cfg.cfg`** – config file used by Configuro
    - **`app.h`** – a header file that app.c depends on
    - **`COPY_AND_PASTE.mak`** – where you will copy/paste some items from
    - **`makefile`** – the makefile you copied from the previous lab
    - **`../../setpaths.mak`** – this file specifies all of the tools paths; it's located two levels above your current working directory

33. **Open `app.c` and study its contents.**

    `app.c` contains a header file (`app.h`) that provides us with the "year" – just a little concoction to use a header file. Also, notice the use of `System_printf()` and the include of the runtime system header file. Close `app.c`.

34. **Open `app_cfg.cfg`.**

    This is the config file used by *Configuro*. Notice that it has one line of code that uses the `xdc.useModule` to specify the module and package we want to consume. Close `app_cfg.cfg`.

35. **Open `app.h`.**

    This simple header file creates an integer variable for the current year (`int YYYY`) which prints into stdout when we run the application.

36. Open `setpaths.mak` and browse the contents.

Migrate up two levels to the `labs` directory. Open `setpaths.mak` and browse the contents. Notice all of the specific path names for all of the tools. This is a similar file that you will need in your application – although some paths may need to change depending on your configuration.

What is the variable name of the path to the Linux 86 gcc compiler? ____________________

What is the variable name of the path where the Linux 86 tools are installed? ___________

Close the file and return back to `lab05c_x86_configuro/app` directory.

37. Add `setpaths.mak` and CC_ROOT to your makefile.

Near the top of your makefile, change the `–include` to the following:

```
–include ../../setpaths.mak
```

Remove or comment out the reference to `path.mak`.

Configuro will need the ROOT path to where the Linux 86 tools are installed. Under the heading for "*user-defined vars*", add the following variable:

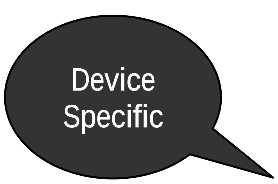
```
CC_ROOT := $(LINUX86_DIR)
```

38. Add the *Configuro* variables to your makefile.

Open `COPY_AND_PASTE.mak` file in your favorite editor and also open your `makefile` in the same editor. In `COPY_AND_PASTE.mak`, find the first comment field for "*Configuro vars*". Copy this whole section (including the comments) and paste it into your `makefile` just beneath the section titled "*User-defined Vars*".

Let's briefly review what each of these variables are used for:

- `CONFIG` : output directory for files generated by Configuro, e.g. compiler.opt: also, used to specify part of .cfg filename
- `XDCROOT` : root directory for where XDC tools are installed
- `CONFIGURO` : location of the Configuro tool
- `XDCPATH` : path containing all packages we want to consume; export makes this variable available to commands run in the shell, for example, Configuro



- `TARGET` : specifies the target, e.g. Linux86 in this case
- `PLATFORM` : specifies the platform – in our case, the PC – later it will be the DM6446, OMAP, or AM3517 target boards

39. Add rule to delete implicit compilation rules.

Copy *"deletion of implicit rules for object file"* section and paste it into your `makefile` just beneath "*Configuro Vars*" (what you just copied in the previous step.)

The code we're copying is:

```
%.o : %.c
```

When this code is used by itself, it erases the previously defined rule for building .o files. It seems that make won't read our .o rule correctly (further down in the file) if we don't erase the implicit rule for it.

Sidebar – Implicit Rules

From gnu.org:

> *Implicit rules* tell `make` how to use customary techniques so that you do not have to specify them in detail when you want to use them.

Since we have created our own customized .o rule, we don't want a conflict with the implicit rule. In fact, many users so dislike implicit rules that they cancel them all. The method we used here works well for cancelling a rule or two, but to eliminate them all, the '`-r`' or '`--no-builtin-rules`' option cancels all predefined rules.

40. Add `.PRECIOUS` directive to prevent removal of intermediate files.

By default, *gMake* will remove intermediate files it uses during the build process. Well, *Configuro* creates `compiler.opt` and `linker.cmd` files and places them in a directory. We don't want *gMake* to erase these files (because we might want to inspect them later).

`.PRECIOUS` directive tells *gMake* NOT to remove these files. In `COPY_AND_PASTE.mak`, copy the section named "*always keep these intermediate files*" and paste it into your `makefile` just beneath "*deletion of implicit rules for object file*".

41. Add `linker.cmd` and `compiler.opt` to the *.x* and *.o* rules.

Configuro creates two files: `compiler.opt` and `linker.cmd` as inputs to the compiler and linker respectively. These files need to be added to the `.x` and `.o` rules along with the `$(CONFIG)` directory (that's where *Configuro* put them).

In the `.x` rule of your `makefile`, add the following dependency:

```
$(CONFIG)/linker.cmd
```

In the `.o` rule, add the following dependency:

```
$(CONFIG)/compiler.opt
```

Also, in the `.o` rule, just before the "`-c`" on the command line, we need to add the following:

```
$(shell cat $(CONFIG)/compiler.opt)
```

This command places the contents of `compiler.opt` on the command line.

42. A little <u>quiz</u> to keep things interesting (and to break the flow a little…)

Study the `.o` rule for a moment. Look at the command that contains `$(CC)`. Just after the –c on this line, you should see a `$<` to indicate first dependency only. And, if you use `$^` to indicate both dependencies, *gMake* will fail. Explain:

____

____

Now look at the `.x` rule. Study the command that contains `$(CC)`. Notice that this time we use `$^` (or you should have from before based on the discussion material) to indicate both dependencies. If you use `$<`, *gMake* will not produce the proper output. Explain:

____

____

43. Add the Configuro Rule.

Well, we're almost done. We now need to add the rule for *Configuro* to create the `linker.cmd` and `compiler.opt` files based on the input file `app_cfg.cfg`.

In `COPY_AND_PASTE.mak`, copy the section named "*Configuro Rule (.cfg)*" and paste it into your `makefile` just above the clean rule.

Let's examine what each line of code does:

```
%/linker.cmd %/compiler.opt : %.cfg
```

There are two targets in this rule – `linker.cmd` and `compiler.opt` (which will be located in the `/app_cfg` directory). These targets depend on a config file (`.cfg`). The pattern substitution symbol (`%`) is used to represent "`app_cfg`".

The command line of this rule runs the *Configuro* tool with all of the necessary inputs as described in the discussion material.

The last little rule (`%.cfg`) is there just in case a `.cfg` file is missing. If so, *gMake* would crash. So, if it doesn't exist, we create an empty file so *gMake* won't crash. Your output won't work, but at least *gMake* won't bomb.

Add one more step to the "clean" rule to remove *Configuro's* intermediate files.

In your `makefile`, add the following command to your clean rule:

```
rm –rf  $(CONFIG)
```

44. Time to test your new makefile.

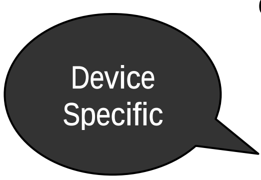
Run *gMake* by typing: `make`

You might see a warning of some kind – just ignore this for now. Run the executable. Did it work? If not, debug your problem and re-build/run.

The only two other rules are "clean" and "test". Try them both.

45. What other functions are in the system package?

Open the header file to see what other functions are provided in the system package:




```
$(XDC_INSTALL_DIR)/packages/xdc/runtime/System.h
```

Where XDC_INSTALL_DIR is:

```
DM6446:  /home/user/dvsdk_2_00_00_22/xdctools_3_10_03
OMAP35x: /home/user/ti-dvsdk_omap3530-evm_4_00_00_17/xdctools_3_16_03_36
AM3517:  /home/user/ti-dvsdk_omap3530-evm_4_00_00_17/xdctools_3_16_03_36
```

As many of you experienced programmers already know, the appropriate header file is a good place to find this type of information.

Page left intentionally blank.

Lab05d_standard_make

Part D – Analyzing TI's Standard Makefile

The authors of this workshop have developed a "one size fits all" makefile for generating executables for the rest of the workshop. Of course, if you adopt this makefile back at work, you might have to change paths (in setpaths.mak), or alter some of the options (such as the targets or platforms) all depending on what you're doing. However, this solution is a pretty robust.

46. Introducing the <u>parent</u> and <u>child</u> makefiles.

We have actually developed a set of two makefiles (the parent – called `makefile`; and the child – called `makefile_profile.mak`). Here are just a few highlights of the overall capabilities of these makefiles:

- They can build using two different profiles: *debug* and *release*
- These makefiles build for the ARM target on the EVM. An install rule exists that automatically copies the executables to the proper directory on the EVM so that you can run via the Tera Term terminal.
- Full "clean" rule is provided.
- They handle dependencies (i.e. header files) from all .c files and any consumed packages.
- The parent takes the input from the command line and invokes the child with the proper profile and settings.
- There are also a few debug features built in to help find make script errors.
- The child does most all of the work **-** dependencies, configuro, .x and .o rules.

In this section (Part D), we only cover the use of these files. The next section (Part E – Challenge) encourages you to open up these files and learn more about their mechanics – but only if time permits.

47. Change to the `lab05d_standard_make/app` directory and list the files.

Everything should look very similar – same `.c` and `.h` files, `app_cfg.cfg`, etc. However, there are two makefiles: `makefile` is the PARENT; `makefile_profile.mak` is the CHILD. When you run *make*, `makefile` calls `makefile_profile.mak`. The main reason for having two files is to handle different profiles – *debug* and *release*. Otherwise, there would be a ton of duplicated code.

```
cd ~/labs/lab05d_standard_make/app
```

48. Let's see some of the features of these make files by running them.

Let's start out easy and just make the *debug* profile:

```
make debug
```

Watch the screen. There is a LOT of information NOT being displayed. By designing the files the way we did, we tried to make the output look simple and uncluttered. We'll see how to turn on ALL the info in a few steps. List the contents of the directory again. Do you see **`app_debug.xv5T`**? If so, make worked. *(To repeat ourselves once again, in the next (optional) section – time permitting – you will open these files and browse their contents.)*

Device Specific

Note: Our executable programs use file extensions to differentiate between different target processors. While Linux doesn't require file extensions, this is a convenient way to allow several to co-exist, as well as just simply tell them apart.

```
.x86U      - Linux x86
.x470MV    - DM6446 ARM9 (using MontaVista toolchain)
.xv5T      - OMAP35x/AM35x (using Code Sourcery toolchain)
```

---

49. Perform a "make clean" and observe the messages on the screen.

50. Using "help".

The authors built in some "help" information. Try:

```
make help
```

Peruse what just flashed before your eyes. These tips help you understand HOW to run this make file properly.

51. Make "all".

Device Specific

Type:

```
make all
```

The all rule builds both the *release* and *debug* versions of the application. When gMake is done, you should see two executables: `app_debug.xv5T` and `app_release.xv5T`. You can't run these on an x86 PC, but next we will install them to the EVM so that we can run them to test if they are working properly.

52. Make "install".

Run a "make clean" first, then try:

```
make install
```

Executing just the *install* rule will automatically create the debug version of the application and install them to `/opt/workshop` directory on the EVM (which is `/home/user/targetfs/opt/workshop` within Ubuntu Linux). If you don't have a terminal open, open a terminal to the EVM using Tera Term. Log in as "root" and change to the `/opt/workshop` directory. Do you see the two executables?

53. Run the debug executable.

Device Specific

Verify the debug executable works. You will need a "`./`" in front of the filename for the target board's *Linux* to recognize the filenames.

```
ll  (lowercase LL – is an alias for ls –l)

./app_debug.xv5T
```

54. Let's turn on some debug stuff…

The parent `makefile` allows you to specify debugging commands on the command line. Let's try two of the built-in "TELL ME EVERYTHING" switches.

First, do a "make clean". Then, to allow gMake to echo each command it is asked to execute set "AT=" nothing on the command line:

```
make clean
make debug AT=
```

Looks different, eh? Well, when (and if) you should NEED to view that information, this trick overrides the AT variable which is normally set to @.

Do another "clean". There is also a "DUMP" switch that will output what each variable is set to (using gMake's *$(warning)* function), along with some other debug information. Trying it:

```
make clean
make debug DUMP=1
```

55. Check to verify that the *dependencies* rule is working correctly.

To verify the dependencies rule is working, first ensure everything is up-to-date by building with *debug* once again; then touch `app.h`, then try building again. If it runs the compiler the second time, it's working properly.

```
make debug
touch app.h
make debug
```

Did gMake run *gcc* after `app.h` was touched (i.e. changed)? ____________________________

____

Lab05e_optional_challenge

Part E – OPTIONAL – CHALLENGE – Analyzing the Details of the Makefiles

This optional lab takes you through some of the details of the two makefiles. At some point, if you decide to use these makefiles for your own builds, you'll need the information below. There are also some excellent references online to help you learn more about gMake.

Some great resources are:

http://www.gnu.org/software/make/manual/make.html

http://www.delorie.com/gnu/docs/make/make_toc.html

www.nso.edu/general/computing/TeX/local/texinfo/gmake/Top.html

And there are many many more – just Google gmake and see what pops up.

56. Browse the contents of the parent makefile: **`lab05d_standard_make/app/makefile`.**

Open `makefile` with a text editor:

```
gedit makefile &
```

We decided to use two makefiles to handle different profiles – these being "debug" or "release". If we only used one file to handle both profiles (and you could have more profiles than just two), you would end up repeating many of the rules and commands for each profile. So, instead of repeating this code over and over, we chose to let the parent (`makefile`) to call the child (`makefile_profile.mak`) with the appropriate profile setting. Thus, the parent makefile formats the users request, then passes it onto the child makefile which contains the script to execute the detailed commands.

You'll also notice that the parent makefile contains a lot of echos/warnings to provide *help* as well as make gMake progress look clean and useful. You may or may not like the fancy syntax – and can change it to suit your needs if you apply it to your own projects back home.

Let's take a brief look at the parent makefile (named `makefile`) – from top to bottom.

A. The `AT` variable helps us turn on/off echos from *gMake*. The default is to NOT echo all the commands that *gMake* spits out. You can leave this as is for a cleaner output – you can change it to "`AT :=` " in makefile – or, on the command line, use "`make debug AT =` " to change its value. As you go down into the file, you'll see how "`AT`" is used.

B. gMake's filter function determines if you added "install" on the a command line, if so, then it's passed to the child makefile via the $(INSTALL) variable.

C. Being the 1st rule found, the "all" rule runs if no target is specified on the command line.

D. If no targets are specified along with "install", we build both *debug* and *release* profiles.

E. Under the "Rules" heading, look at `debug` and `release`. We use the –f to call the child makefile (`makefile_profile.mak`), the INSTALL variable, and the profile (debug or release).

F. The "`clean`" rule sends the child the "`clean`" goal along with the profile.

G. The rest of the file contains the "`help`" rule – that tells you how to use this makefile.

makefile   (i.e. "parent" makefile)

Step 56a

Debug variable AT. By default, it's set to "@".

It is used to prevent commands being echoed

Step 56b

If "install" is specified on the command line, then we set the INSTALL variable, otherwise leave it blank.

Step 56c

"all" rule – first in line makes it the default rule

Step 56d

If only "install" is specified as a target, then both debug and release are built (similar to call "all install".

Step 56e and f

Step 56g

```
# ------------------------------------------------------------------------------
# makefile     Version 1.0        Date: SEPT-29-2008              (edited to fit this page)
#
# Use:
# - this is the PARENT makefile to makefile_profile.mak. You can specify any
#   child and run gMake with the proper options.
# ------------------------------------------------------------------------------

# AT: Used for debug purposes, it hides commands for a prettier output.
#     When debugging, you can set this to nothing on the make command line.
# ---------------------------------------------------------------------
AT := @

# ---------    INSTALL: See description from 'help' below    ----------
# ---------------------------------------------------------------------
ifeq ($(filter install,$(MAKECMDGOALS)),install)
 INSTALL := install
else
 INSTALL :=
endif

# -----------------                Rules
# -----------------------------------------
.PHONY  : all debug release clean install help

all     : debug release

ifeq ($(MAKECMDGOALS),install)
 install        : debug release
                  @echo "Install was called without other targets, so both 'debug' and 'release' were built"
else
 install        :
                  @echo
endif

debug  :
                  $(AT) make -f makefile_profile.mak $(INSTALL) PROFILE=DEBUG | grep –v -F make[1]
                  @echo "Done building 'debug'"   ; echo
release :
                  $(AT) make -f makefile_profile.mak $(INSTALL) PROFILE=RELEASE  | grep -v -F make[1]
                  @echo "Done building 'release'" ; echo
clean    :
                  @ echo "---- Cleaning up files for $(firstword $(MAKEFILE_LIST)) -----"
                  $(AT) make -f makefile_profile.mak clean PROFILE=DEBUG   | grep -v -F make[1]
                  $(AT) make -f makefile_profile.mak clean PROFILE=RELEASE | grep -v -F make[1]

help         :
                  @echo "This makefile serves as a 'parent' (or master) makefile. That is, it calls another makefile
                  @echo "called 'makefile_profile.mak'. If the child makefile is called directly, it will build only
                  @echo "one profile (by default, it builds the 'DEBUG' profile). This parent makefile allows
                  @echo "you to easily build for multiple profiles with a single invocation.
                  @echo
                  @echo "The goals allowed by this makefile are:  all, debug, release, clean, install, help
                  @echo
                  @echo "  debug:  calls the child makefile with the "DEBUG" profile "
                  @echo "release:  calls the child makefile with the "RELEASE" profile "
                  @echo "    all:  calls the child makefile twice, once with "DEBUG", then with "RELEASE"
                  @echo "  clean:  calls the child makefile twice to clean both debug and release "
                  @echo "install:  adds the 'install' goal to the child makefile's target, then calls child. Install"
                  @echo "          will make BOTH profiles (release and debug) and install them to the DVEVM "
                  @echo
                  @echo "To DUMP additonal makefile variables, use 'DUMP=1' when you run make."
```

Note:

The makefile shown in print is the one included with the DM6446 lab exercises. There are a few minor differences in the OMAP/AM35 makefiles.

Overall, the parent simply handles the profiles and calls the child based on the goals listed when you invoke make. The child really does all the work to build the executables.

57. Open the child (`makefile_profile.mak`) which is called by the parent (`makefile`).

`Makefile_profile.mak` builds for the ARM9 target – however, other targets could easily be supported (with a little tweaking). The parent makefile passes the "PROFILE" (*release* and/or *debug*) and "INSTALL" variables to the child make file which performs the appropriate commands based on these parameters. All dependencies (e.g. header files) are handled by the dependency rule. The child uses Configuro to consume packages delivered by TI or 3rd parties (similar to how you wrote a previous part of this lab). All tools paths are specified in `setpaths.mak`, which is located in the `labs` directory (two levels above `app`).

In the following steps, we'll look at the main pieces of the child makefile to understand how it works. We'll do this chronologically from the top of the file to the bottom. Not every piece will be covered in detail, so referencing the links provided earlier may help you understand gMake even better.

```
gedit makefile_profile.mak &
```

58. "Early" Include file – `setpaths.mak`.

Near the top of `profile_makefile.mak`, you'll notice we included `setpaths.mak`. If you don't remember what is contained in this file, feel free to open it up and view its contents.

Files are included in two spots in this make script: early and late. In our case, we need the paths defined early on, otherwise a number of references would fail.

Conversely, if we include dependency (.d) files right away, that generates an error; therefore, we include these towards the end of the file.

59. User-defined variables – for the Compiler.

Under the comment banner*"User-defined Variables"*, you'll see the standard variable types that we used earlier, but notice that there are now two versions of compiler flags:

- debug (e.g. DEBUG_CFLAGS)
- release (e.g. RELEASE_CFLAGS)

Again, the parent passes the value of $(PROFILE) to the child at which point it's the child's responsibility is to build the executable program. You'll notice we need two sets of CFLAGS – one for each profile.

The standard CFLAGS and LINKER_FLAGS variables have been modified to appropriate flags needed to build ARM9 programs.

Note: If `makefile_profile.mak` was called without defining PROFILE, then it defaults to *debug*. A little later in this file we actually set PROFILE:=DEBUG to defines its default value.

---

makefile_profile.mak (i.e. "child" makefile)

```
# ******************************************************************************
#
# makefile_profile.mak     Version 1.0       Date: SEPT-29-2008
#
# Revisions:
# - (v0.80) First "standard" GNU makefile for DaVinci Workshop
#           used in workshop versions 1.30 (beta's 1 & 2)
# - (v1.00) Used for DaVinci Workshop (production version 1.30)
#
# Use:
# - Called by parent makefile named "makefile"
# - Can be called directly using gMake's -f option; refer to the syntax used
#   by the "parent" makefile to invoke this make file
# - Currently builds for ARM9 target, however other targets can be supported.
# - User can specify PROFILE (either debug
#   or release or all) when invoking the parent makefile
# - All dependencies (e.g. header files) are handled by the dependency rule
# - Uses Configuro to consume packages delivered by TI
# - All tools paths are specified in setpaths.mak located two levels above /app
#
# ******************************************************************************

# *****************************************************************************
#
#    (Early) Include files
#
# *****************************************************************************

# ---------------------------------------------------------------------
# setpaths.mak includes all absolute paths for DaVinci tools
#   and is located two levels above the /app directory.
# ---------------------------------------------------------------------
-include ../../setpaths.mak

# *****************************************************************************
#
#    User-defined vars
#
# *****************************************************************************

# ---------------------------------------------------------------------
# AT: - Used for debug, it hides commands for a prettier output
#     - When debugging, you may want to set this variable to nothing by
#       setting it to "" below, or on the command line
# ---------------------------------------------------------------------
AT := @

# ---------------------------------------------------------------------
# Location and build option flags for gcc build tools
#   - MONTAVISTA_DEVKIT is defined in setpaths.mak
#   - CC_ROOT is passed to configuro for building with gcc
#   - CC is used to invoke gcc compiler
#   - CFLAGS, LINKER_FLAGS are generic gcc build options
#   - DEBUG/RELEASE FLAGS are profile specific options
# ---------------------------------------------------------------------
CC_ROOT :=  $(MONTAVISTA_DEVKIT)/arm/v5t_le
CC      :=  $(CC_ROOT)/bin/arm_v5t_le-gcc

CFLAGS       := -Wall -fno-strict-aliasing -march=armv5t -D_REENTRANT \
                -I$(MONTAVISTA_DEVKIT)/arm/v5t_le/include
LINKER_FLAGS := -lpthread

DEBUG_CFLAGS   := -g -D_DEBUG_
RELEASE_CFLAGS := -O2
```

Step 58

Need to include our tool paths. They are defined in setpaths.mak and located such that this file is common for all labs.

Step 59

Along with the build flags for ARM9 gcc, we see there are two sets of flags associated with our "profile" choices (debug, and release).

60. **Creating arrays of C, object and dependency files.**

    Inspect the five lines of code that start with C_SRCS. The goal here is to create an array of object files and dependency files based on the existing C files in the current directory. So, we first create an array of .c files (C_SRCS) using gMake's wildcard function. Once we have this array, we can create a corresponding array of object files – C_OBJS – use two additional gMake functions (*subst* and *addprefix*). Similarly, we also use C_SRCS to create the array of dependency files – C_DEPS.

    Note, you'll see these variables being used further down in the child makefile.

61. **Inspecting the Configuro variables.**

    The next section should look familiar. You either wrote or copied this code in a previous part of this lab. To review, these are the variables that will be used in the Configuro rule later in the child makefile.

62. **Project specific variables.**

    Rather than hardcoding the program name, configuration filename, and profile, they have been created as variables. This should make it easier to adapt the makefile's for other programs/projects.

63. **Understanding PRECIOUS.**

    Scroll down a small amount and find the directive .PRECIOUS. This might be new to you, so let's explain it briefly. gMake, by default, deletes intermediate files unless you tell gMake not to. So, for instance if file1.c is used to build file2.o which is used in the final step to build file3.x470MV, then gMake may delete file2.o UNLESS you tell it not to. In our case, we don't want gMake to remove the C_OBJS array or the linker.cmd and compiler.opt files that Configuro creates. So, we use the .PRECIOUS directive to say "please DO NOT delete these files".

64. **Deleting implicit rules for object files.**

    gMake has implicit rules – i.e. if you don't tell it exactly what to do, it performs its own implicit rules. You could create a makefile with no rules, or rules with no commands, etc. So, we are just being a bit conservative here and telling gMake NOT to use any implicit rules for .o files. If you want to learn more about implicit rules, commands, etc., refer to the links provided earlier.

makefile_profile.mak (cont'd 2)

Step 60

Array of source files to build; plus the object and dependency files derived from the C files in this directory

Step 61

Configuro related variables: 2 tool paths; search path; and, target/platform def's

Step 62

Name of program to build

Name of Configuro .cfg file

Default profile name

Step 63 is Precious

Step 64

FWIW, we like explicit rules

```
# -------------------------------------------------------------------
# C_SRCS used to build two arrays:
#   - C_OBJS is used as dependencies for executable build rule
#   - C_DEPS is '-included' below; .d files are build in rule #3 below
#
# Three functions are used to create these arrays
#   - Wildcard
#   - Substitution
#   - Add prefix
# -------------------------------------------------------------------
C_SRCS := $(wildcard *.c)

OBJS   := $(subst .c,.o,$(C_SRCS))
C_OBJS  = $(addprefix $(PROFILE)/,$(OBJS))

DEPS   := $(subst .c,.d,$(C_SRCS))
C_DEPS  = $(addprefix $(PROFILE)/,$(DEPS))

# -------------------------------------------------------------------
# Configuro related variables
# ---------------------------
#  - XDCROOT is defined in setpaths.mak
#  - CONFIGURO is where the XDC configuration tool is located
#  - Configuro searches for packages (i.e. smart libraries) along the
#    path specified in XDCPATH; it is exported so that it's available
#    when Configuro runs
#  - Configuro requires that the TARGET and PLATFORM are specified
#  - Here are some additional target/platform choices
#      TARGET   := ti.targets.C64
#      PLATFORM := ti.platforms.evmEM6446
#      TARGET   := gnu.targets.Linux86
#      PLATFORM := host.platforms.PC
# -------------------------------------------------------------------
XDCROOT := $(XDC_INSTALL_DIR)
CONFIGURO := $(XDCROOT)/xs xdc.tools.configuro
export XDCPATH:=/home/user/rtsc_primer/examples;$(XDCROOT)
TARGET   := gnu.targets.MVArm9
PLATFORM := ti.platforms.evmDM6446

# -------------------------------------------------------------------
# Project related variables
# -------------------------
#   PROGNAME defines the name of the program to be built
#   CONFIG:  - defines the name of the configuration file
#            - the actual config file name would be $(CONFIG).cfg
#            - also defines the name of the folder Configuro outputs to
#   PROFILE: - defines which set of build flags to use (debug or release)
#            - output files are put into a $(PROFILE) subdirectory
#            - set to "debug" by default; override via the command line
# -------------------------------------------------------------------
PROGNAME := app
CONFIG   := app_cfg
PROFILE  := DEBUG

# -------------------------------------------------
# ----- always keep these intermediate files ------
# -------------------------------------------------
.PRECIOUS : $(C_OBJS)
.PRECIOUS : $(PROFILE)/$(CONFIG)/linker.cmd $(PROFILE)/$(CONFIG)/compiler.opt

# -------------------------------------------------
# --- delete the implicit rules for object files --
# -------------------------------------------------
%.o : %.c
```

65. Default rule.

Being the first rule listed in the file, *Default_Rule* becomes the, ahem, default rule. Notice, this rule depends upon the ARM executable program we really want to build.

When this rule runs, it generates a single, empty line (from the *echo* command). Therefore, even when all the dependencies are up-to-date and nothing needs to be built, this makefile will generate at least one blank line. This may seem like an odd point, but the parent makefile would show an error if there wasn't anything written to stdio.

66. Build executable.

The next rule builds the final executable – either the DEBUG profile, the RELEASE profile. This part should look pretty familiar to you based on previous sections of this lab.

Notice the use of PROFILE as a variable. If we're building both *debug* and *release*, we'll do this rule twice in order to build both executables. (That is, both can be built, but only by the parent `makefile` calling `makefile_profile.mak` for each profile.)

Having to manage PROFILES (debug and release) is made much easier by using two makefiles. Otherwise, you have a lot of duplicate code in a single makefile. (Actually, our first attempt was to do it in one file – and it was VERY long – intimidating – so, we decided to have one makefile call another – and, in a way, it taught us the concept of multiple – recursive – makefiles.)

As a side-note, we found it helpful to see a "count down" in the build output to the finish. What does that mean? In the echo statements, you'll see a "1. ----- …", "2. ---- …" etc, that provides an indication of how far gMake still has to go until it is finished. So, the last step – building the executable – is actually "1". The first step is actually "4". So, when you build using these makefiles, you'll see the echo statements reflect 4…3…2…1… and then it finishes. This is not necessary for the build – it just makes the information output to the stdout window easier to read.

67. Object File Rule.

The .o rule should also look familiar. Nothing new here except for the PROFILE variable.

The PROFILE variable here represents the subfolder we are placing our intermediate files into. This is done so that we don't overwrite our *debug* variables when building *release*, and vice-versa.

The key to understanding this `target:dependency` rule is to *follow the %*:

```
$(PROFILE)/%.o : %.c $(PROFILE)/$(CONFIG)/compiler.opt
```

That is, just remember that **%** represents a substitution symbol. So, if I have a source file named:

```
bar.c
```

Then (when building for *debug*) I'll end up with a target object file named:

```
DEBUG/bar.o
```

While this might be obvious so many of you, it's a common question we get asked regarding this rule.

makefile_profile.mak (cont'd 3)

```
# ***************************************************************************
#
#    Targets and Build Rules
#
# ***************************************************************************

# -----------------------------------------------------------------------
# Default Rule
# ------------
#  - When called by the "parent" makefile, being the first rule in this
#    file, this rule always runs
#  - Depends upon ARM executable program
#  - Echo's linefeed when complete; this target was added to
#    prevent the parent makefile from generating an error if the
#    ARM executable is already built and nothing needs to be done
# -----------------------------------------------------------------------
Default_Rule : $(PROGNAME)_$(PROFILE).x470MV
         @echo

# -----------------------------------------------------------------------
# 1. Build Executable Rule  (.x)
# ----------------------------
#  - For reading convenience, we called this rule #1
#  - The actual ARM executable to be built
#  - Built using the object files compiled from all the C files in
#    the current directory
#  - linker.cmd is the other dependency, built by Configuro
# -----------------------------------------------------------------------
$(PROGNAME)_$(PROFILE).x470MV : $(C_OBJS) $(PROFILE)/$(CONFIG)/linker.cmd
         @echo; echo "1.  ----- Need to generate executable file: $@ "
         $(AT) $(CC) $(CFLAGS) $(LINKER_FLAGS) $^ -o $@
         @echo "Successfully created executable : $@ "

# -----------------------------------------------------------------------
# 2. Object File Rule (.o)
# ------------------------
#  - This was called rule #2
#  - Pattern matching rule builds .o file from it's associated .c file
#  - Since .o file is placed in $(PROFILE) directory, the rule includes
#    a command to make the directory, just in case it doesn't exist
#  - Unlike the TI DSP Compiler, gcc does not accept build options via
#    a file; therefore, the options created by Configuro (in .opt file)
#    must be included into the build command via the shell's 'cat' command
# -----------------------------------------------------------------------
$(PROFILE)/%.o : %.c $(PROFILE)/$(CONFIG)/compiler.opt
         @echo "2.  ----- Need to generate: $@ (due to: $(wordlist 1,1,$?) ...)"
         $(AT) mkdir -p $(dir $@)
         $(AT) $(CC) $(CFLAGS) $($(PROFILE)_CFLAGS) \
                     $(shell cat $(PROFILE)/$(CONFIG)/compiler.opt) -c $< -o $@
         @echo "Successfully created:  $@ "
```

Step 65

Default_Rule is the default rule … wait do I hear an @echo here?

Step 66

An executable rule, similar to early parts of this lab.

We've only added *profile* (i.e. path) variables (and a few comments) to the .x rule

Step 67

Similar to the .x rule, we've added profile/path vars to the .o rule

68. Handling C File Dependencies.

This part of the makefile may be new to you. We discussed it in the chapter, but not in full detail.

This rule uses the compiler to create a dependency (**.d** for dependency) file which corresponds to each .c file in the current directory. What does the .d file contain? A list of dependencies (i.e. header files) referenced by the .c file.

These .d files helps gMake do what it's good at, trigger the rule to run if any of the dependent files are newer than the target. It is common to miss including header files as dependencies for .c targets; using the compile to generate this information is a great solution to the problem.

The `-MM` gcc option is used to tell the compiler to capture this dependency information, rather than compiling the file. We still provide it the same flags and files, though, just as if we were compiling the file.

In our rule, we pipe the outputs of the `gcc –MM` command into a file. We then format the compiler's output using a gMake macro (*format_d*). We adapted a set of commands – found on various gMake related websites – that reformat this list of dependency files into a gMake rule. For example, `app.o` depends on `app.h` (along with any other header file listed in `app.c`).

When gMake runs these rules, it checks the dates on the header files to see if any are newer than the corresponding .o file.

In the *format_d* macro (found near the bottom of the file), you'll see its command uses a string "reformatting" tool – **sed** – which stands for "stream editor". Sed is a convenient – albeit cryptic - way to process text strings.

69. Config Rule(s).

The Configuro rule should look familiar. Except for the PROFILE path, this should be nearly what you added to your makefile in a previous part of the lab to run Configuro; and thus, consume a package (e.g. for consuming the *system_printf()* function in `app.c`) .

The only other change we made was to alter the information output when running Configuro. We have made the Configuro output into a sort of quiet mode by piping them into a log file. If, on the other hand, you want to see this information, you can set DUMP=1 on the command line and all Configuro's verbosity will be displayed.

Finally, we added one last Configuro related command to prevent an error in the case where our specified .cfg file doesn't exist. The following command:

```
touch $(CONFIG).cfg
```

prevents this error condition. If Configuro attempts to run without a .cfg file, an error causes gMake to stop. So, when/if that case occurs, we create an empty config file using *touch*. This shouldn't hurt anything (unless you just forgot to provide the .cfg file), because Configuro fires up, sees that you haven't included any packaged content, and exits. Since we did not specify .cfg files as PRECIOUS, this temporary, intermediate file is deleted by gMake (as per it's standard operating procedure).

makefile_profile.mak (cont'd 4)

Step 68

We're depending on this rule to make sure no changes are missed

```
# -------------------------------------------------------------------------
# 3. Dependency Rule (.d)
# -----------------------
#  - Called rule #3 since it runs between rules 2 and 4
#  - Created by the gcc compiler when using the -MM option
#  - Lists all files that the .c file depends upon; most often, these
#    are the header files #included into the .c file
#  - Once again, we make the subdirectory it will be written to, just
#    in case it doesn't already exist
#  - For ease of use, the output of the -MM option is piped into the
#    .d file, then formatted, and finally included (along with all
#    the .d files) into this make script
#  - We put the formatting commands into a make file macro, which is
#    found towards the end of this file
# -------------------------------------------------------------------------
$(PROFILE)/%.d : %.c $(PROFILE)/$(CONFIG)/compiler.opt
	@echo "3.  ----- Need to generate dep info for:        $< "
	@echo "          Generating dependency file   :        $@ "
	$(AT) mkdir -p $(PROFILE)
	$(AT) $(CC) -MM $(CFLAGS) $($(PROFILE)_CFLAGS) \
	                $(shell cat $(PROFILE)/$(CONFIG)/compiler.opt) $<  > $@

	@echo "Formatting dependency file: $@ "
	$(AT) $(call format_d ,$@,$(PROFILE)/)
	@echo "Dependency file successfully created: $@ " ; echo

# -------------------------------------------------------------------------
# 4. Configuro Rule (.cfg)
# ------------------------
#  - The TI configuro tool can read (i.e. consume) RTSC packages
#  - Many TI and 3rd Party libraries are packaged as Real Time Software
#    Components (RTSC) - which includes metadata along with the library
#  - To improve readability of this scripts feedback, the Configuro's
#    feedback is piped into a a results log file
#  - In the case where no .cfg file exists, this script makes an empty
#    one using the shell's 'touch' command; in the case where this
#    occurs, gMake will delete the file when the build is complete as
#    is the case for all intermediate build files (note, we used the
#    precious command earlier to keep certain intermediate files from
#    being removed - this allows us to review them after the build)
# -------------------------------------------------------------------------
$(PROFILE)/%/linker.cmd $(PROFILE)/%/compiler.opt : %.cfg
	@echo "4. -- Starting Configuro for $^  (note, this may take a minute)"
    ifdef DUMP
	$(AT) $(CONFIGURO) -c $(CC_ROOT) -t $(TARGET) -p $(PLATFORM) \
	                   -r $(PROFILE) -o $(PROFILE)/$(CONFIG) $<
    else
	$(AT) mkdir -p $(PROFILE)/$(CONFIG)
	$(AT) $(CONFIGURO) -c $(CC_ROOT) -t $(TARGET) -p $(PLATFORM) \
	                   -r $(PROFILE) -o $(PROFILE)/$(CONFIG) $<  \
	                                    > $(PROFILE)/$(CONFIG)_results.log
    endif
	@echo "Configuro has completed; it's results are in $(CONFIG) " ; echo

# -------------------------------------------------------------------------
# The "no" .cfg rule
# -----------------
#  - This additional rule creates an empty config file if one doesn't
#    already exist
#  - See the Configuro rule comments above for more details
# -------------------------------------------------------------------------
%.cfg :
	$(AT) touch $(CONFIG).cfg
```

Step 69

Configuro, Configuro, where art thou Configuro

The ifdef DUMP is used to 'quiet' its output

The touch command prevents an error for programs that don't need a .cfg file (i.e. aren't using RTSC packaged content)

70. **Build, clean and Install.**

    Nothing here should surprise you. We could’ve just phoned this in.

    All we did here was to add echo’s to provide a bit more feedback during build.

makefile_profile.mak (cont'd 5)

```
# **************************************************************************
#
#    "Phony" Rules
#
# **************************************************************************

# ----------------------------------------------------------------------
#  "all" Rule
# -----------
#   - Provided in case the a user calls the commonly found "all" target
#  - Called a Phony rule since the target (i.e. "all") doesn't exist
#    and shouldn't be searched for by gMake
# ----------------------------------------------------------------------
.PHONY  : all
all     : $(PROGNAME)_$(PROFILE).x470MV
         @echo ; echo "The target ($<) has been built."
         @echo


# ----------------------------------------------------------------------
#  "clean" Rule
# -------------
#  - Cleans all files associated with the $(PROFILE) specified above or
#    via the command line
#  - Cleans the associated files in the containing folder, as well as
#    the ARM executable files copied by the "install" rule
#  - EXEC_DIR is specified in the included 'setpaths.mak' file
#  - Called a Phony rule since the target (i.e. "clean") doesn't exist
#    and shouldn't be searched for by gMake
# ----------------------------------------------------------------------
.PHONY : clean
clean  :
         @echo ; echo "--------- Cleaning up files for $(PROFILE) -----"
         rm -rf $(PROFILE)
         rm -rf $(PROGNAME)_$(PROFILE).x470MV
         rm -rf $(EXEC_DIR)/$(PROGNAME)_$(PROFILE).x470MV
         rm -rf $(C_DEPS)
         rm -rf $(C_OBJS)
         @echo


# ----------------------------------------------------------------------
#  "install" Rule
# ---------------
#  - The install target is a common name for the rule used to copy the
#    executable file from the build directory, to the location it is
#    to be executed from
#  - Once again, a phony rule since we don't have an actual target file
#    named 'install' -- so, we don't want gMake searching for one
#  - This rule depends upon the ARM executable file (what we need to
#    copy), therefore, it is the rule's dependency
#  - We make the execute directory just in case it doesn't already
#    exist (otherwise we might get an error)
#  - EXEC_DIR is specified in the included 'setpaths.mak' file; in our
#    target system (i.e. the DVEVM board), we will use /opt/workshop as
#    the directory we'll run our programs from
# ----------------------------------------------------------------------
.PHONY  : install
install : $(PROGNAME)_$(PROFILE).x470MV
         @echo
         @echo  "0. -- Install $(PROGNAME)_$(PROFILE).x470MV to 'Exec Dir' --"
         @echo  "          Execution Directory:  $(EXEC_DIR)"
         $(AT) mkdir -p $(EXEC_DIR)
         $(AT) cp    $^ $(EXEC_DIR)
         @echo  "          Install (i.e. copy) has completed" ; echo
```

all, clean, install

A common set of phony rules.

What does .PHONY mean? Only that these target names don't represent real filenames. Phony just tells gMake not to go looking for any files named: all, clean, or install.

71. Macro: format_d

As stated before, this macro reformats the list of dependency files (created by running the compiler with the –MM option) into gMake rules. This allows make to verify that the dependent files (i.e. header file) timestamps are not later than the .o files created from the .c files that reference them. (Whew, that's a mouthful.) In other words, when a header file gets modified, you want the object (.o) file to be rebuilt.

Here, as we've seen elsewhere, we use the DUMP variable to inject additional debugging information. If DUMP exists, then we embed a *$(warning)* function into each .d file; this warning shouts out whenever the .d file is read by gMake. You probably won't need this, but it helped us track down a bug or two.

72. Including C_DEPS.

We include our .d files at the end of our make script, rather than the beginning. If we included them at the same time that `setpaths.mak` was included, we would receive an error; this error happens because we are including the array of files specified by C_DEPS, but that variable wasn't defined before `setpaths.mak` was included. Therefore, we've put it at the end of our make file.

As we've seen elsewhere gMake supports *ifeq*/*endif* conditional statements. The conditional statement says, include all the .d files unless the MAKE GOALS include *clean*. (We don't need the dependency files when cleaning, as our *clean* rule doesn't delete source files.)

Since make will try to read the .d files on the first pass, before be build any of the targets, the first time this include is run will likely result in an error. We can tell make to ignore this error by using the "-" symbol.

```
include foo      # don't ignore an error
-include foo     # ignore an error if it occurs when running this command
```

An odd, but handy aspect of gMake is that when an *included* file is updated during its execution, it forces gMake to re-run the entire make script over from the beginning. So, even if we get an (ignored) error the first time we run this *include*, once the .d files are created (by our .d rule), the make file will be re-executed and our *include* should work this time around.

One last little item to point out. The command:

```
-include $(C_DEPS)
```

is run recursively. If you were to look back how C_DEPS was defined, you'll notice we used "=" rather than ":=". This tells make we want this to be a *recursive* variable. This include statement is a perfect example of why we want this. In most cases C_DEPS will hold a string of filenames, e.g. "`app.d foo.d … bar.d`". Due to the nature of recursive variables, our single include command will end up acting like:

```
-include app.d
-include foo.d
…
-include bar.d
```

Pretty darn handy, huh?

makefile_profile.mak (cont'd 6)

Step 71

If we sed it before, we'll say it again. We created this macro to encapsulate the formatting of the file dependency info spit out by gcc's –MM option.

Sure, we could've just put these lines of script straight into our .d rule, but: (1) it would have looked messier; and (2) we wouldn't have had the chance to try out gMake macros …

Step 72

Lately, we've been making some pretty mean .d files.

Seriously, here's how our .d files get included into our build.

If we're cleaning, we're not going to include them.

No worries if the .d files don't exist by the time we execute this statement. This is actually to be expected. So, by adding the "-" before include, we're just telling make to ignore any errors caused by our "–include"

```
# ****************************************************************************
#
#    Macros
#
# ****************************************************************************
# format_d
# --------
#  - This macro is called by the Dependency (.d) file rule (rule #3)
#  - The macro copies the dependency information into a temp file,
#    then reformats the data via SED commands
#  - Two variations of the rule are provided
#     (a) If DUMP was specified on the command line (and thus exists),
#         then a warning command is embed into the top of the .d file;
#         this warning just lets us know when/if this .d file is read
#     (b) If DUMP doesn't exist, then we build the .d file without
#         the extra make file debug information
# -------------------------------------------------------------------
ifdef DUMP
  define format_d
   @# echo " Formatting dependency file: $@ "
   @# echo " This macro has two parameters: "
   @# echo "   Dependency File (.d): $1     "
   @# echo "   Profile: $2                  "
   @mv -f $1 $1.tmp
   @echo '$$(warning --- Reading from included file: $1 ---)' > $1
   @sed -e 's|.*:|$2$*.o:|' < $1.tmp >> $1
   @rm -f $1.tmp
  endef
else
  define format_d
   @# echo " Formatting dependency file: $@ "
   @# echo " This macro has two parameters: "
   @# echo "   Dependency File (.d): $1     "
   @# echo "   Profile: $2                  "
   @mv -f $1 $1.tmp
   @sed -e 's|.*:|$2$*.o:|' < $1.tmp > $1
   @rm -f $1.tmp
  endef
endif


# ****************************************************************************
#
#    (Late) Include files
#
# ****************************************************************************
#  Include dependency files
# -------------------------
#  - Only include the dependency (.d) files if "clean" is not specified
#    as a target -- this avoids an unnecessary warning from gMake
#  - C_DEPS, which was created near the top of this script, includes a
#    .d file for every .c file in the project folder
#  - With C_DEPS being defined recursively via the "=" operator, this
#    command iterates over the entire array of .d files
# -------------------------------------------------------------------
ifneq ($(filter clean,$(MAKECMDGOALS)),clean)
  -include $(C_DEPS)
endif
```

makefile_profile.mak (cont'd 7)

```
# ****************************************************************************
#
#    Additional Debug Information
#
# ****************************************************************************
#  Prints out build & variable definitions
# ----------------------------------------
#  - While not exhaustive, these commands print out a number of
#    variables created by gMake, or within this script
#  - Can be useful information when debugging script errors
#  - As described in the 2nd warning below, set DUMP=1 on the command
#    line to have this debug info printed out for you
#  - The $(warning ) gMake function is used for this rule; this allows
#    almost anything to be printed out - in our case, variables
# ---------------------------------------------------------------------

ifdef DUMP
  $(warning To view build commands, invoke make with argument 'AT= ')
  $(warning To view build variables, invoke make with 'DUMP=1')

  $(warning Source Files: $(C_SRCS))
  $(warning Object Files: $(C_OBJS))
  $(warning Depend Files: $(C_DEPS))

  $(warning Base program name : $(PROGNAME))
  $(warning Configuration file: $(CONFIG))
  $(warning Make Goals        : $(MAKECMDGOALS))

  $(warning Xdcpath :  $(XDCPATH))
  $(warning Target  :  $(TARGET))
  $(warning Platform:  $(PLATFORM))
endif
```

73. Print out build information.

In the last part of the child make file, you'll see a bunch of *$(warning)* statements. This is a handy way to print out some information on gMake variables, which could make debugging make easier. Looking at the file, you'll see these warnings will only show up if you have "DUMP=1" on the command line. (Alternatively, you could add the DUMP variable to the make file itself, but since we shouldn't need to debug this file anymore, defaulting to off is probably better.)

Lab 6 - Using the ALSA Driver

Introduction

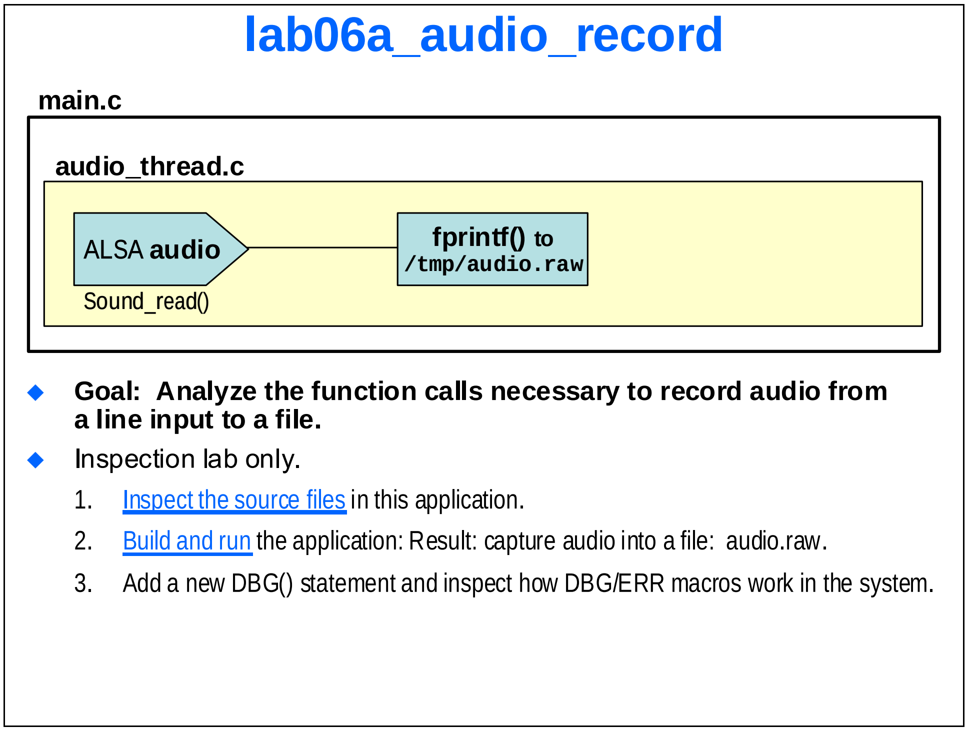
In Lab 6, we will inspect the first two labs (recorder and playback) and then stitch the input driver to the output driver to create the loopthru application.

The labs demonstrate the Linux ALSA driver as well as basic file I/O. Labs 06a and 06b are inspection labs. While Lab06c requires you to combine the previous two parts.

- **Lab06a** analyze the function calls necessary to record audio from line input to a file.
- **Lab06b** examines the function calls necessary to playback audio from a recorded audio file.
- **Lab06c** combines Lab06a and Lab06b into a single application that loops the audio from input to output (i.e. audio loop-thru) without recording to a file. *For an extra challenge, advanced users may want to try to build lab06c without referring to the procedure.*

Outline

Lab06a_audio_record



File Management

1. In VMware (Ubuntu), change to the following directory:

`/home/user/labs/lab06a_audio_record/app`

2. List the files used to build this application:

. ____

. ____

. ____

. ____

File Inspection

3. Use a text editor to examine the new files in this application.

A number of text editors are available to you in Linux. You should use what you are comfortable with. Probably the most user-friendly is **gedit**, invoked as:

```
gedit main.c
```

Other popular editors are emacs and gvim.

main.c

This is the entry point for the application. ***main()*** does the following:

- Creates a signal handler to trap the Ctrl-C signal (also called `SIGINT`, the interrupt signal). When this signal is sent to the application, the *audioEnv.quit* global variable is set to true to signal the audio thread to exit its main loop and begin cleanup.
- Calls the ***audio_thread_fxn()*** function to enter into the audio function.
- Upon completion of this function, the main routine checks – and reports – success or failure returned from the audio function.

audio_thread.c

audio_thread_fxn()

audio_thread_fxn( ) encapsulates the code required to run the audio recorder. The `lab06a_audio_recorder` application is single-threaded, so the motivation for encapsulation in this manner may not be initially obvious. We will see in labs 8a and 8b – when combining audio and video in a multi-threaded program – why declaring this function (as opposed to running everything from main) is useful.

audio_thread_fxn utilizes the following:

- ***setup:*** DMAI method *Sound_create()* opens and configures the audio input driver. *Buffer_create* allocates a RAM buffer to store the audio input, *fopen()* opens a file `(audio.raw)` for recording.
- ***System():*** In the thread create phase, you will see two system() calls that run amixer. When you first use DMAI and do a Sound_create, you would think that this function would turn on the LINE IN inputs. Well, after a week or so of struggling, the author figured out that "you won't get audio unless you run these two system commands". So, this is one piece that is NOT done by DMAI for you – there, you are forewarned. ☺
- ***while():*** will execute until the *envPtr->quit* global variable is set to true.
  - Inside the *while()* loop, *Sound_read()* is used to read data from the audio input driver (ALSA) and *fwrite()* is used to write the data into a file.
  - When the *envPtr->quit* variable is set to true (occurs when the user presses Ctrl-C in the terminal) this capture (record) process exits and the application proceeds to the cleanup phase before exiting.

initMask

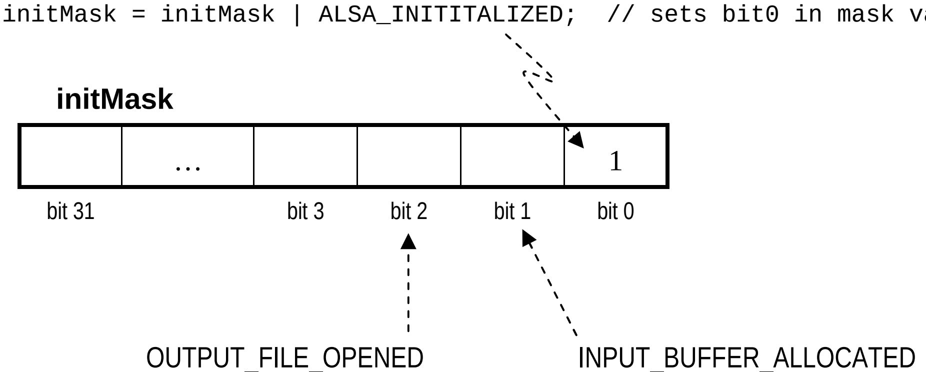
It goes without saying, writing robust code – and debugging it – can be a tedious chore; it is further exasperated when using printf() statements as the primary means of providing debug information back to the programmer. To this end, we have employed an *initMask* to help keep track of resources opened (and closed) during the program.

The *audio_thread_fxn()* uses an initialization mask (*initMask*) to keep track of how many resources have been opened and initialized. Each bit in the mask corresponds to a resource; the bit positions in the *initMask* variable are `#defined` towards the top of the file.

```
/*  The levels of initialization for initMask */
#define ALSA_INITIALIZED        0x1
#define INPUT_BUFFER_ALLOCATED  0x2
#define OUTPUT_FILE_OPENED      0x4

/* Only used to cleanup items that were initialized */
   unsigned int initMask = 0x0;
```

When you OR the initMask with a #define'd value, the associated bit will get set in the initMask variable. For example,

```
initMask = initMask | ALSA_INITITALIZED;  // sets bit0 in mask variable
```



This is useful so that if an error occurs, the application will not attempt to close or free resources that were never opened or allocated. If you look down at the *cleanup* part of our `audio_thread.c`, you'll see how we used the initMask variable to accomplish this.

app_cfg.cfg

This is the eXpress DSP Component (XDC) tool configuration file for the application. It imports the Operating System Abstraction Layer (OSAL) module (required by DMAI) and configures it for a Linux-only system.

Build and Run the application

4. **Build and install the application using gMake, i.e.** `“make all install”`**.**

   Make sure you are in the `/app` directory when you type “make…”.

5. **Open a terminal to the EVM board. Log in and use the following terminal commands to test your audio connection:**

```
# amixer cset name=’Analog Left AUXL Capture Switch’ 1

# amixer cset name=’Analog Right AUXR Capture Switch’ 1

# arecord –f cd | aplay –f cd
```

   The first two commands use the amixer (“ALSA mixer”) utility to enable left and right channel capture. The final command uses the arecord (“ALSA recorder”) utility to capture audio and, instead of sending to a file, uses a Linux process pipe to send the data to the aplay (“ALSA player”) application. This will loop audio through the board. If you have a working audio input and the board is connected to a speaker, you should hear the audio play over the speakers. Press ctrl-c to quit.

   On `arecord` and `aplay`, the `–f` option lets you change the format:

Quality	# Channels	Bits	Rate
default	mono	8-bit	8 KHZ
cd	stereo	16-bit	44.1 KHz

6. **Execute `loadmodules.sh`.**

   Navigate to `/opt/workshop` on the EVM board.

   Remember, loadmodules is a script that will insert some additional modules into the Linux kernel – for example, CMEM.

7. **Execute the `./app_DEBUG.xv5T` application.**

   The application is hard-coded (using a `#define` statement in `audio_thread.c`) to save the audio data to the file `/tmp/audio.raw`.

   Execute the application.

8. **Press Ctrl-C to exit the application.**

   After a suitable amount of time press Ctrl-C in the terminal to exit from the application. You can list the `/tmp/audio.raw` file with the *–lsa* options to see the size of the file and verify that it has recorded properly:

```
ls –lsa /tmp/audio.raw
```

   Recall that a signal handler was placed in `main.c` to trap the `SIGINT` (Ctrl-C) signal. When Ctrl-C is placed, this signal handler will execute, signaling the audio thread to exit its main loop, proceed to cleanup, and then exit.

9. Use the Linux aplay utility to confirm successful recording.

Because this application saves the audio as a raw stream you may also check that the record has operated properly using the aplay utility. On the EVM (i.e. in Tera Term):

```
#  aplay -c 2 -f S16_LE -r 44100 /tmp/audio.raw
```

DBG vs ERR

Let's explore the debugging features we're using in our lab files. We are using two macros defined in the file `debug.h`. They are *DBG()* and *ERR()* – essentially, they are wrapper functions around an *fprintf()* function.

10. Add a new debug statement to your file.

In `main.c`, immediately after the signal handler function, add a DBG() statement:

```
// Set the signal callback for Ctrl-C
signal( SIGINT, signal_handler );
DBG( "Registered SIGINT signal handler.\n" );
```

11. Build (using "`make all install`") and run both *debug* and *release* profiles on the development board (EVM), comparing their outputs.

Does your new statement show up in the terminal when you execute the program?

Debug profile: ❑ Yes ❑ No ______________________________

Release profile: ❑ Yes ❑ No ______________________________

12. Switch from DBG() to ERR(), then once again, build, run and compare both profiles.

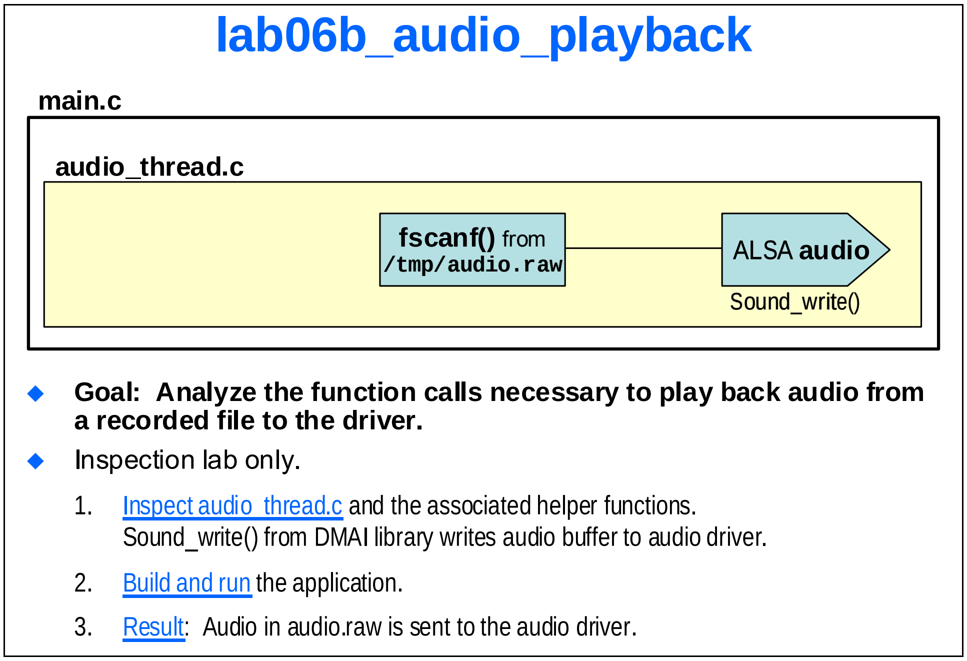
What is the difference between DBG and ERR? ______________________________

____

13. Either Delete the new ERR() statement, or switch it back to DBG().

We don't really need this statement, so feel free to remove it. On the other hand, if you want to leave the new debugging statement, we recommend that you, at the very least, change it back to a DBG() statement.

Lab06b_audio_playback



File Inspection

audio_thread.c

14. In Ubuntu Linux (VMware PC), change to the directory:

```
/home/user/labs/lab06b_audio_playback/app
```

15. Use a text editor to examine `audio_thread.c`.

Only `audio_thread.c` has changed from the `lab06a_audio_record` application. The other files are unchanged. Let’s look at some of the differences:

- The ‘create’ part of the *audio_thread_fxn()* utilizes *DMAI* to:
  - Uses the *fopen()* function call to open a file for playback.
  - *Sound_create* and configure the audio output driver.
  - The *Buffer_create()* function allocates a RAM buffer to store the audio data from the input file before it is written to the audio driver.
- Inside the *while()* loop:
  - *fread()* method is used to read audio data from the input file (`/tmp/audio.raw`)
  - *Sound_write()* method is used to write the data to the ALSA driver.
  - When the *envPtr->quit* variable is set to true, the loop exits. (This occurs when the user presses Ctrl-C in the terminal.)
- Finally, review the “cleanup” phase, which runs right before exiting. (This basically undo’s the steps in the create/setup phase).

Build and Run the Application

16. Build and install the application using gMake.

```
make debug install
```

or

```
make install
```

17. Make sure that `audio.raw` was created properly.

Navigate to `/opt/workshop` in the EVM board's filesystem and list the contents of the `/tmp` directory with the "-lsa" flags setting to verify that `/tmp/audio.raw` exists and has a greater than zero filesize.

The application is hard coded (using a #define statement in `audio_thread.c`) to read data from the file `/tmp/audio.raw`. Note that the `/tmp directory` is located in the board's RAM. (All other directories reside on the host computer and are tied to the board's filesystem via the *nfs* file-sharing protocol.)

Note: If the EVM is reset or powered-off after running the `lab6a_audio_record` application, the `/tmp/audio.raw` file will be erased from RAM memory.

If this happens, run "make install" again from the `lab06a_audio_record` directory to re-install the audio recorder. Then run it once in either debug or release mode to re-record the `/tmp/audio.raw` file

Finally, you can return to `lab06b_audio_playback` and run gMake to re-install the playback utility.

---

18. Execute the `./app_DEBUG.xv5T` application.

The application should play back the audio that was recorded in `lab06a_audio_record` and then exit. If you do not wish to hear all of the audio, press Ctrl-C to exit.

Questions about: `audio_thread.c`

1. Which C structure (variable type and variable name) is used to set the audio input driver to *stereo, 44100 kHz* using the ALSA driver?

   . ____

2. Which function call is used in the *while()* loop to read audio data from the EVM line input via the ALSA driver? ____

   . ____

3. In the *while( )* loop there is an *fread()* function (similar to *fwrite()* function, lab06a).

   For the file read (or write) function, a FILE pointer is the last parameter passed. What is the purpose of the FILE pointer, and where does it come from? (In other words, what function is used to generate valid FILE pointers from which read and write operations can be made?)

   . ____

   . ____

4. (Advanced) The *Buffer_create()* function call, by default, allocates a physically contiguous DDR2 memory buffer. This is not the behavior of a standard *malloc()* call – which only allocates virtually contiguous memory.

   Which Linux module is used to provide the physically contiguous memory segments allocated by this function call?

   . ____

   . ____

5. (Advanced) The *Buffer_create()* function call uses the above-mentioned Linux module to allocate contiguous memory on a Linux system. On a DSP/BIOS system, the Buffer_create() function would use DSP/BIOS *MEM_alloc()* function call instead of the aforementioned Linux function.

   How does the *Buffer_create()* function call know which of these function calls to make? *(Hint: How was the DMAI module imported into this project? Try looking in that file.)*

   . ____

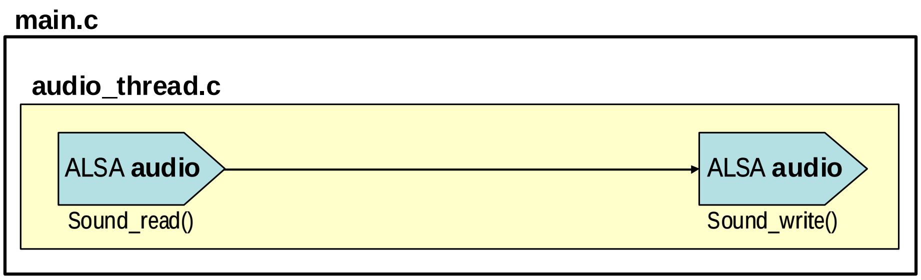
   . ____

   . *(Note: While DSP/BIOS is not covered in this class, we added this question to show both the versatality of the Codec Engine, as well as the ease-of-use configurability of RTSC modules).*

Lab06c_audio_loopthru

In this lab, you will combine labs 06a and 06b into a single loopthru application. For an extra challenge, advanced students may wish to see if they can accomplish this lab without referring to the procedure.

lab06c_audio_loopthru



- **Goal: Combine the record (lab06a) and playback (lab06b) into an audio loopthru application.**
- Hey – YOU get to do this yourself (no more inspection stuff…)
  1. Answer a few questions about the big picture (covered in the next few slides…)
  2. Copy files from lab06b (playback) to lab06c (loopthru)
  3. Make code modifications to stitch the record to the playback (covered in the next few slides…).
  4. Build, run. Result: audio is recorded (from ALSA input), copied from in ? out buffer, then played back (to ALSA output).

What do we need to change?

Before we start copying, cutting, and pasting files and code, let's think about what must be done to get the loopthru lab to work.

- In `Lab06a_audio_record`, we used ***fwrite()*** to PUT (write) the audio data to the `audio.raw` file. Which function was used to GET (read) the video data from the ALSA driver?

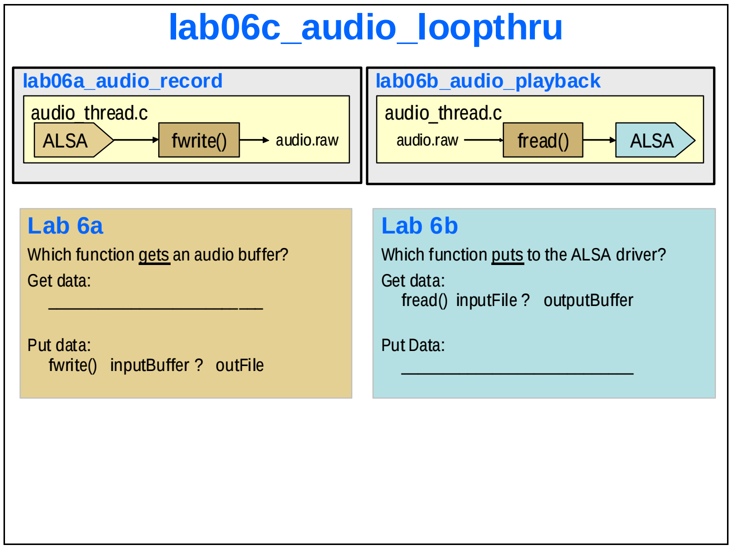
```
GET audio data:    ________________________________________

PUT audio data:   fwrite()  inputBuffer -> audio.raw ________
```

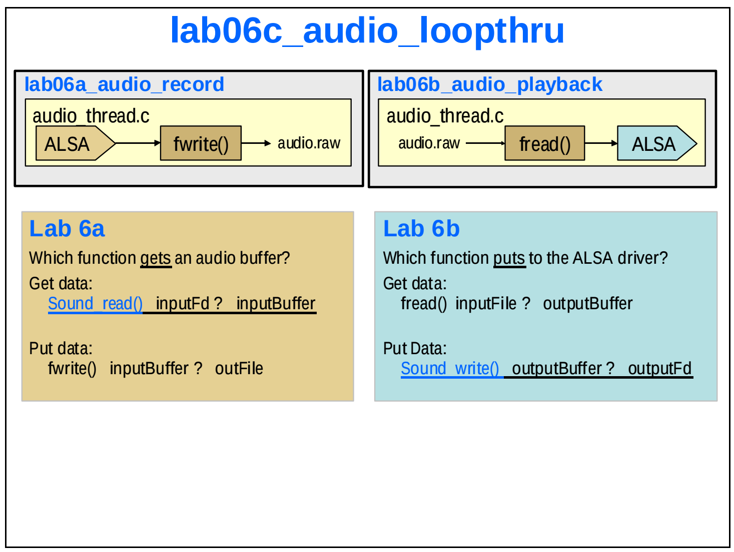
Similarly, in `Lab06b_audio_playback`, we used the function listed below to PUT (write) the data to the ALSA driver. Which function was used to GET (read) the audio data?

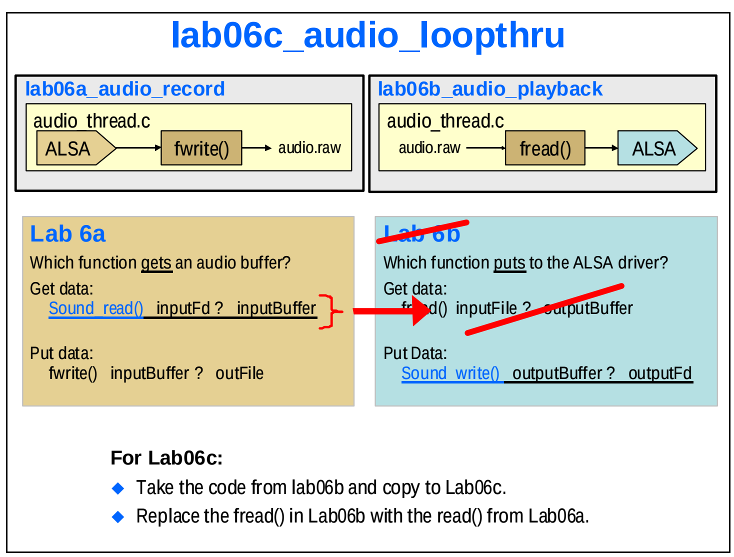
```
GET audio data:    ________________________________________

PUT audio data:   Sound_write() outputBuffer -> hSound ______
```

- In this lab exercise, which two functions should be used to read/write data to the input/output ALSA driver?

```
Get audio data:    ________________________________________

Put audio data:    ________________________________________
```


lab06c_audio_loopthru
lab06a_audio_record
audio_thread.c
ALSA
fwrite()
audio.raw
lab06b_audio_playback
audio_thread.c
audio.raw
fread()
ALSA
Lab 6a
Which function gets an audio buffer?
Get data:
Put data:
fwrite() inputBuffer ? outFile
Lab 6b
Which function puts to the ALSA driver?
Get data:
fread() inputFile ? outputBuffer
Put Data:


lab06c_audio_loopthru
lab06a_audio_record
audio_thread.c
ALSA
fwrite()
audio.raw
lab06b_audio_playback
audio_thread.c
audio.raw
fread()
ALSA
Lab 6a
Which function gets an audio buffer?
Get data:
Sound_read() inputFd ? inputBuffer
Put data:
fwrite() inputBuffer ? outFile
Lab 6b
Which function puts to the ALSA driver?
Get data:
fread() inputFile ? outputBuffer
Put Data:
Sound_write() outputBuffer ? outputFd


lab06c_audio_loopthru
lab06a_audio_record
audio_thread.c
ALSA
fwrite()
audio.raw
lab06b_audio_playback
audio_thread.c
audio.raw
fread()
ALSA
Lab 6a
Which function gets an audio buffer?
Get data:
Sound_read() inputFd ? inputBuffer
Put data:
fwrite() inputBuffer ? outFile
Lab 6b
Which function puts to the ALSA driver?
Get data:
fread() inputFile ? outputBuffer
Put Data:
Sound_write() outputBuffer ? outputFd
For Lab06c:
Take the code from lab06b and copy to Lab06c.
Replace the fread() in Lab06b with the read() from Lab06a.

File Management

18. Begin by copying all files from `lab06b_audio_playback` **into** `lab06c_audio_loopthru` **with the following:**

```
cd /home/user/labs
mkdir –p lab06c_audio_loopthru
cp –R –f lab06b_audio_playback/* lab06c_audio_loopthru
```

The ***mkdir*** "–p" option prevents an error if the directory already exists.

The ***cp*** "-R" options says to recurse directories, while the "-f" option forces over write if the file already exists.

Note: Since lab06c is a combination of lab06a and lab06b (and only file that differs between them is `audio_thread.c`) , you could have copied the other directory over first, then made changes to it. But, even so, we highly recommend you follow the directions above so that the next steps are consistent with your files/directories.

Modify audio_thread.c

19. Open `audio_thread.c` in a text editor.

Navigate to `lab06c_audio_loopthru/app/`.

Numerous editors are available to you depending on what you are most comfortable with. Some common options are:

```
gedit audio_thread.c &
```

20. Begin by removing the #define statement that sets INPUTFILE.

While not absolutely necessary, we might as well clean up anything that will not be used later, and removing it here will help us catch any errors if we forget to remove some file-related commands. (In the place of the #define INPUTFILE, in the next step we'll add the proper command needed for the ALSA driver.)

21. In *audio_thread_fxn()* in the declarations, modify the `#define` bit settings for the initMask to have three initialization states:

- ALSA_DRIVER_INITIALIZED
- INPUT_BUFFER_ALLOCATED
- OUTPUT_BUFFER_ALLOCATED

It doesn't matter which bit you allocate to each, as long as they are each independent bits in the mask, i.e. 0x1, 0x2, 0x4, 0x8, etc.

22. Remove the input file initialization code

- Remove the following code section in ***audio_thread_fxn()*** of `audio_thread.c`:

```
/* Open input file */
...
```

  And this one, too:

```
/* Record that input file was opened in initialization bitmask */
...
```

23. Modify the *mode* field of the sound device attributes to enable bidirectional operation.

Before editing the **`sAttrs.mode`** variable, let's first check what options are available in DMAI:

The easiest way to do this is to look directly inside of the DMAI header file: **`Sound.h`**

DVSDK 4.0: **`cd /home/user/ti-dvsdk_omap3530-evm_4_00_00_17/dmai_2_05_00_18`**

then: **`gedit packages/ti/sdo/dmai/Sound.h`**

Within this file you will find the `Sound_Mode` enumerated type definition (around line 95). What is the enumeration value for bidirectional operation (i.e. full duplex)?

. Bidirectional operation mode enumeration: ____

Finally, edit the variable in your program:

```
sAttrs.mode =
```

24. Declare a new `Buffer_Handle` *`hBufIn`* and initialize it to `NULL`.

Since we're doing a pass through application we could get by with a single buffer; reading into it using *Sound_read()*, then writing it back out with *Sound_write()*.

However, when we add our audio processing to this thread, it will be helpful to have separate input and output buffers. So, in this lab we'll go ahead and create a separate input buffer. Add the new buffer handle next to the one already allocated for `hBufOut`.

```
Buffer_Handle hBufIn
```

(Note: In place of audio processing – which we'll add in a later lab exercise – in step 28 we'll use the *memcpy()* function to copy the data from the input buffer to the output buffer.)

25. Allocate an audio input buffer.

Because we copied lab06b (audio playback) into lab06c, we already have a section of code that creates the audio output buffer. Inspect the code after the following banner:

```
/* Initialize the output audio buffer */
```

We still need this code. However, we need to add the INPUT side and INPUT audio buffer.

The simplest way to do this is to copy this section of code (thru and including setting the initMask bit) from lab06a. When pasting, place it BEFORE the output section and modify it appropriately for the INPUT audio buffer.

`Copy the entire section from:`

```
/* Initialize the output audio buffer */

…TO…

initMask |= OUTPUT_BUFFER_ALLOCATED;
```

`INCLUSIVE. Then, paste it right ABOVE the output section.`

Change the **`hBufOut`** handle in the copied section to **`hBufIn`** and …

… change the `INITMASK` bitmask to `INPUT_BUFFER_ALLOCATED`.

26. Prime the Pump using *Sound_read()* rather than *fread()*.

You'll find the call that needs to be changed in the "// Prime the Pump" section just before the *while()* loop.

As earlier in step 23, you canmcheck the `Sound.h` header file for the Sound_read() prototype. For convenience, we've reprinted here:

```
Int      Sound_read (Sound_Handle hSound, Buffer_Handle hBuf)
```

Also, don't forget to change the *DBG()* statement that follows to reference `hBufIn`, instead of `inputfile`.

Note: Following the Prime-thePump *Sound_read()*, you'll notice two single writes. These writes avoid any potential underflow condition on the output driver and does not add any noticeable distortion. There is nothing you need to modify here – just FYI.

---

27. Within the *while()* loop, replace *fread()* call with *Sound_read()* call.

If you need a hint, you can reference the *audio_thread_fxn()* in `lab06a_audio_recorder`.

Once again, don't forget to change the *DBG()* statement to use `hBufIn` vs. `inputfile`.

28. Create an audio *pass-thru* using memcpy() to copy data from the input to the output.

You will need to use the Buffer_getUserPtr() function to get the memory pointer associated with each buffer. (The handles hBufIn and hBufOut are pointers to structures, not pointers to the underlying memory buffers).

For example, if you wanted to specify the pointer to the input buffer, you would use:

```
Buffer_getUserPtr(hBufIn)
```

You will also need to use the Buffer_getSize() function to determine how many bytes to transfer. In this case, use the size of the output buffer:

```
Buffer_getSize(hBufOut)
```

The memcpy() prototype is as follows:

```
memcpy(void *write_to, void *read_from, int num_bytes);
```

In your code, substitute the functions provided above into the memcpy call to perform the audio pass thru.

29. Replace the file cleanup code with the Buffer_delete cleanup on hBufIn.

- Locate the *Thread Delete Phase* after the "**cleanup:**" tag in the `audio_thread_fxn` of lab06c. Remove the code section labeled:

  ```
  /* Close input file */
  ```

- Replace the file cleanup code's *fclose()* with the proper *Buffer_delete()* of `hBufIn`.

30. If you opened `audio_thread.c` in `lab06a_audio_record`, you can close it now.

Note, you should not need to save `audio_thread.c` from `lab06a_audio_record` because you should not have modified this file, only copied sections from it to paste into lab06c.

31. Update the variable declarations at the beginning of `audio_thread_fxn`.

After cutting-and-pasting the code in the last few steps, a few new variables have been added and one was removed. Update the variable declarations at the beginning of `audio_thread_fxn`.

The following can be removed:

```
FILE  *inputFile = NULL;
```

Make sure the following variable is declared:

```
Buffer_Handle hBufIn = NULL;
```

32. Save and close audio_thread.c.

Build and Test

33. Build and install the application

```
# make install
```

34. Go to the `/opt/workshop` **folder on the target and run the application.**

You should hear audio playing.

If you have re-booted the board recently and NOT loaded loadmodules.sh on the EVM, this lab will fail to allocate memory. FYI.

Again, if at all possible (and we know it IS possible), please turn down your volume to a reasonable level so as not to disturb your neighbors too much or cause others in the building to call the police and cite you for disturbing the peace. Thanks.

Page left intentionally blank.

Lab 7 - Using Video Drivers

Introduction

This chapter explores the video system drivers. The labs demonstrate the Linux V4L2 and FBdev drivers as well as basic file I/O, through four small applications: on-screen display (OSD), video recorder, video player, and video loop-thru (video-capture copied to video-display).

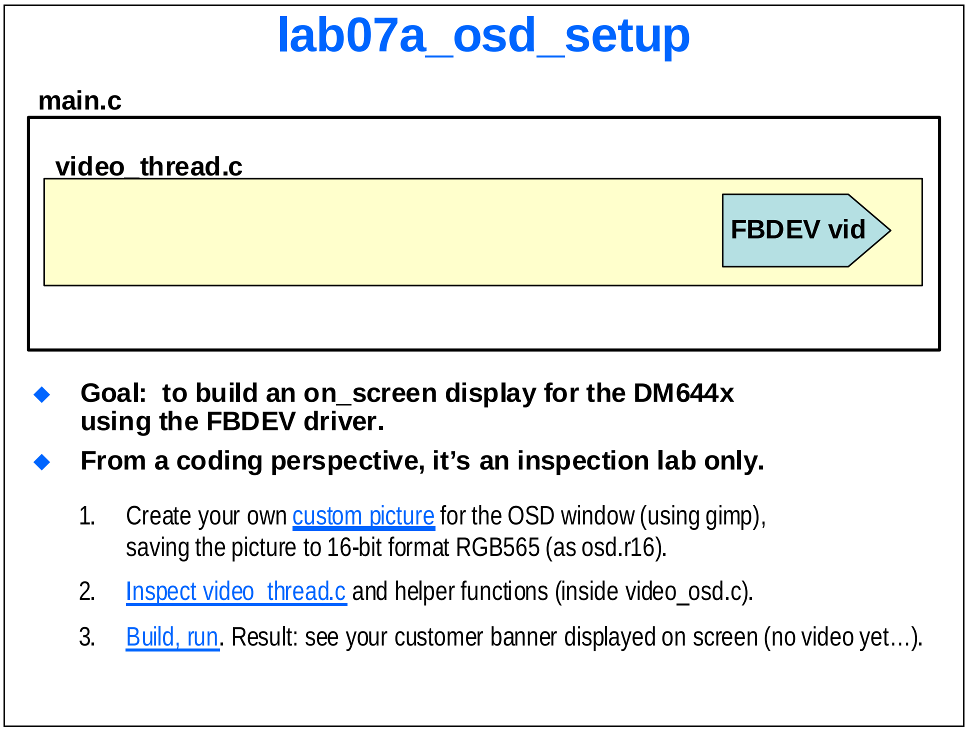
Lab 7 is composed of 4 parts:

- **Lab 07a**: You will build an <u>on-screen display</u> for the your device using the FBDEV (thru DMAI) driver – INSPECTION LAB only.
- **Lab 07b**: Examines v4L2 video capture via a simple <u>video recorder</u> application – INSPECTION LAB only.
- **Lab 07c**: Examines the v4L2 display driver using via a <u>video display</u> application. This application plays back the file captured in lab 07b – INSPECTION LAB only.
- **Lab 07d**: You will combine the recorder and player applications into a <u>video loop-thru</u> application using memcpy to transfer data between capture and display drivers.
- **Lab 07e**: You will modify lab07d to perform video loop-thru via <u>pointer passing</u> between capture and display drivers. (More efficient)

Outline

Lab 7 – Using Video Drivers

Lab07a_osd_setup



Lab 07a Procedure

1. **In Ubuntu Linux, change to the directory:**

   `/home/user/labs/lab07a_osd_setup/osdfiles`

2. **Open the Gimp (open-source) paint program by typing "gimp" in the terminal.**

3. **Create a customer banner picture.**

   Create a new file using: `File` → `New`

   Set the height and width to 80 x 640, since we only want to create a banner, not fill the whole screen. Other than the resolution, nothing else in the new file dialog needs to be modified.

   Width: 640 pixels

   Height: 80 pixels

4. **Paint something for your OSD banner.**

   You can create a simple graphic quickly using just three of the many tools.

   – Before clicking any of the tools, you can choose a color first using the **color box**.

   – Start with the **gradient** tool to create a background. Select the tool, then click and drag the mouse over the 640x80 image area.

   – Add **text** or **paint** something over the gradient with either of these tools, respectively.

Add **text** by selecting the font icon is the capitol T

The **gradient** tool is the box at the top that shows a gradient going from green to white

Paint brush

You can change the colors by left clicking on the **color box**

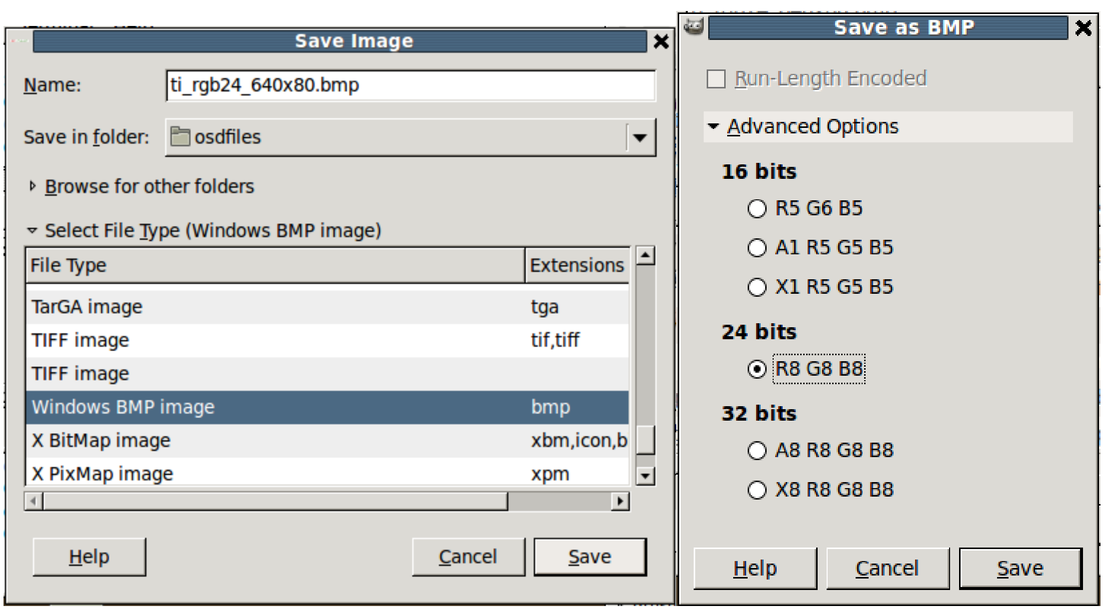
5. **Save your file and exit Gimp.**

   When you are finished, save with **File → Save**. Then exit Gimp.

   Be sure to select "BMP" (bitmap) from the *Select File Type* box. Name your file:

   `ti_rgb24_640x80.bmp`

   It DOES matter what you name the file because later, during the building process, this file is specifically copied to the target. This name also makes it easy to remember it's a 640x80 Bitmap image in 24-bit RGB.



   You will also need to specify the R8 G8 B8 format.

6. **Make sure your file is saved to the `osdfiles` subdirectory in your lab folder.**

7. **Change to the** `lab07a_osd_setup/app` **directory and list the contents.**

8. **Examine two of the video files.**

video_osd.c

`video_osd.c` contains a number of functions for manipulating the on-screen display. Unlike DMAI, these functions are not part of an official package, but were developed for these lab exercises to demonstrate the capabilities of the on-screen display hardware.

- ***video_osd_place()*:** places a picture on the OSD display. Assumes data is provided in 32-bit ARGB (8-bit attribute transparency, 8-bit red, 8-bit blue 8-bit green per pixel).
- ***video_osd_scroll()*:** a more complex version of video_osd_place() that will offset the OSD display by x and/or y scroll values. This can be used to scroll a banner or picture horizontally or vertically.
- ***video_osd_circframe()*:** draws a circular alpha-blended frame around the video output.

osd_thread.c

The thread function in `osd_thread.c` is *`video_thread_fxn()`* which uses the helper functions from video_osd.c as well as an extension of the DMAI Display module (myDisplay) which supports alpha blending:

- calls ***myDisplay_fbdev_create()*** to open the OSD window. This window is memory mapped (mmap'ed) into the application space and a handle to the Display object is returned and stored in hOsd.
- calls ***readPictureBmp()*** to read the custom banner picture (as created in gimp and stored in a 24-bit RGB bitmap file) and store a handle to a buffer object containing the picture into the bannerBuf Buffer handle (which is passed by reference). In addition to reading the 24-bit RGB data from the file, it appends an 8-bit transparency value before each pixel, which in this case is a constant value specified by TRANSP and equal to 0xFF (fully opaque).
- Inside the initialization for() loop, all OSD buffers are initialized by placing the bannerBuf picture buffer using ***video_osd_place(),*** which places the picture on the OSD window with a y offset of 480 (screen height of 480 minus picture height of 80).
- Also inside the initialization for() loop, a call is made to ***video_osd_circframe()*** to initialize all OSD buffers with a circular semi-transparent (0x80 is 50% transparency) blue frame (0x0000FF is blue, 0x00FF00 is green, 0xFF0000 is red).
- Drops into a while loop (testing on *env->quit*, which is changed to 1 when ctrl-C is pressed) in which it scrolls the OSD banner by calling ***myDisplay_fbdev_get*** to gain access to the next OSD buffer, calling video_osd_scroll to update the scrolling buffer, and calling ***myDisplay_fbdev_put*** to display the updated buffer.

The application assumes the picture is supplied in a 640 x 80 24-bit RGB (8-8-8) format, which should be the case if you followed the previous gimp instructions.

9. Build and install the application.

10. Execute the application on the target.

```
target#  cd /opt/workshop
```

Note: loadmodules.sh does not need to be re-executed if you have previously run the script (to load the cmem module into the Linux kernel) since the last reset of the board.

```
target#  ./loadmodules.sh

target#  ./app_debug.xv5T
```

At this point, you should only see a black background with the OSD showing. The OSD should consist of your scrolling graphic along with a circular, semi-transparent frame.

Lab 07a Questions

1. How would you modify the `lab07a_osd_setup` application to make the banner you created semi-transparent instead of solid?

.

.

.

2. How would you modify the `lab07a_osd_setup` application to place your banner at the top of the screen instead of the bottom?

.

.

.

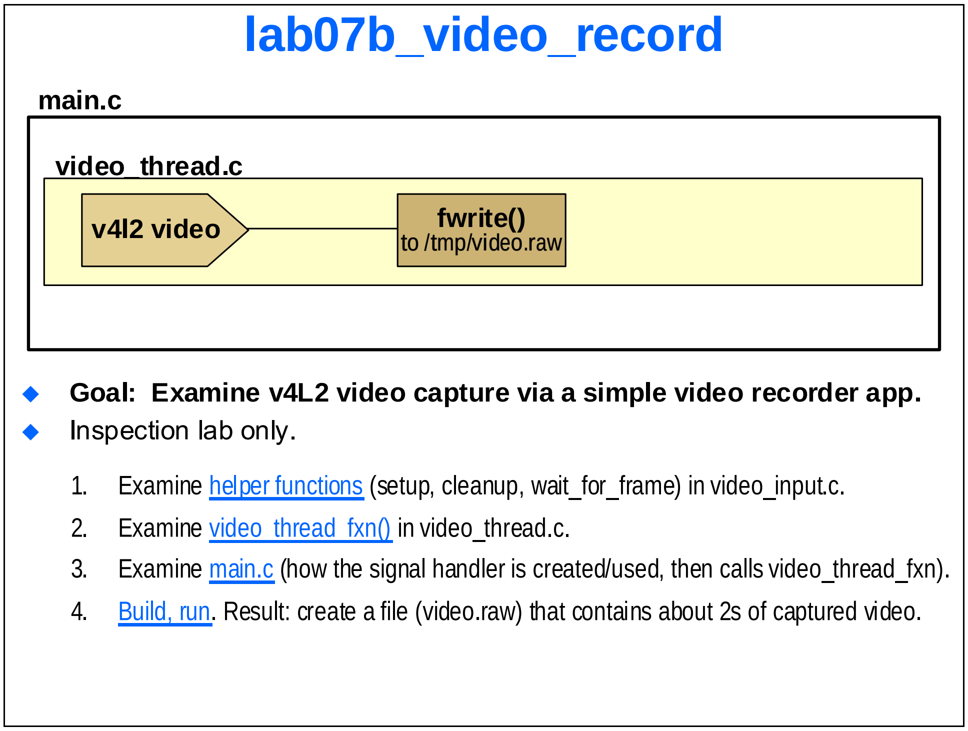
3. (Advanced) Why is the Buffer_Handle bannerBuf preceded with an ampersand (&) in the function readPictureBmp()?

.

.

.

Lab07b_video_capture



11. Change to the directory:

```
/home/user/labs/lab07b_video_capture/app
```

12. Examine the video files:

app_cfg.cfg

As with `lab07a_osd_setup` the eXpress DSP Component (XDC) tool configuration file imports and configures the RTSC-compliant packages used in this application.

These RTSC packages are:

- ***ti.sdo.ce.osal.Global:*** Global Operating System Abstraction Layer (OSAL) module
- ***ti.sdo.dmai:*** The Digital Multimedia Application Interface (DMAI) module
- ***ti.*tto.myDisplay:** An extension of the DMAI Display module for use in these lab exercises

video_thread.c

This file contains a single function, ***video_thread_fxn().*** This function encapsulates the functionality necessary to run the video recorder and is analogous to the ***audio_thread_fxn()*** that was used in lab06.

video_thread_fxn() utilizes the following:

- ***Capture_detectVideoStd() and BufferGfx_calcDimensions():*** these two DMAI functions are used in conjunction to detect the video standard which is input to the development board and calculate the corresponding buffer size of a single video frame.
- ***BufTab_create():*** A method of the BufTab DMAI module which creates a table of Buffers, in this case the video buffers which will be used by the video capture driver to store video frames
- ***Capture_create():*** A method of the Capture DMAI module which opens and configures the Linux V4L2 video capture driver. Since the application passes a bufTab (Buffer Table) handle, the driver will use user-allocated (as opposed to driver-allocated) buffers to store video frames.
- ***fopen():*** Standard Linux I/O (i.e from `#include <stdio.h>`) call to open a file where the captured video data will be written
- ***for()*** loop:
  - *Loops through 10 cycles so as not to overflow /tmp directory's RAM memory*
  - ***Capture_get():*** dequeues the next video frame from the V4L2 driver (blocks/pauses if buffer is not available, yet).

    fwrite(): The video frame is copied into the file.
  - ***Capture_put():*** Once the application has finished writing the video buffer to the file, the buffer handle must be passed back to the driver so that it can be refilled with new video data.

main.c

This is the entry point for the application. ***main()*** does the following:

- Creates a signal handler to trap the Ctrl-C signal (also called `SIGINT`, the interrupt signal). When this signal is sent to the application, the *videoEnv.quit* global variable is set to true to signal the video thread to exit its main loop and begin cleanup.
- After configuring this signal handler, ***main()*** calls the ***video_thread_fxn()*** function to enter into the video thread. Upon completion of this function, ***main()*** checks the return value of the function (success or failure) and reports.

13. Build and install the application using gMake using "`make install`".

14. Run the application on the target

Open a terminal to the EVM board. Navigate to the target's **`/opt/workshop`** directory, and then execute the **`./app_debug.xv5T`** application.

Hint: By the way, make sure you have a video source playing for this lab to look right.

You will get a message from the application indicating that it has recorded 10 captured video frames. Check the following to ensure that the video has recorded properly:

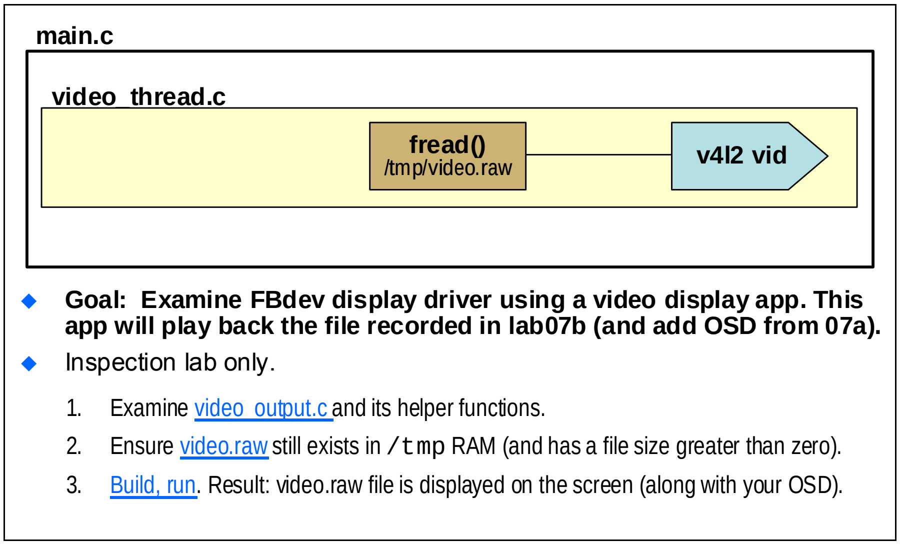
```
ls –lsa /tmp/video.raw
```

The file should be about 7 MB in size. The reason that the application only records ten video frames is to keep from overflowing the `/tmp` directory.

Note: We are saving the file to RAM-based `/tmp` directory because the NFS mounted filesystem that the board is using is too slow to save raw video.

Also note, that you cannot see this file from within Ubuntu because the `/tmp` directory contents are actually stored in RAM, as opposed to on the NFS drive.

Lab07c_video_playback



15. Change to the directory:

`/home/user/labs/lab07c_video_playback/app`

16. Examine `video_thread.c`:

As opposed to the recorder which uses DMAI's *Capture* module, this application uses the *myDisplay* module to display video frames it reads from the `/tmp/video.raw` file:

- ***fopen() and fread():*** Opens the input file containing captured video frames and reads two 4-byte integer values. The first is the *video standard* (as enumerated in DMAI's `videoStd.h`) and the second is the *size of each video frame*.
- ***BufTab_create():*** A method of the *BufTab* DMAI module which, using the `captureSize` variable which was read from the input file, creates a table of appropriately sized buffers to hold video frame data that is read from the file.
- ***Display_create():*** A method of the *Display* DMAI module which opens and configures the Linux V4L2 video display driver. Since the application passes a *bufTab* (Buffer Table) handle, the driver will use user-allocated (as opposed to driver-allocated) buffers to store video frames.
- ***while()*** loop:
  - o ***Loops*** until ctrl-C is pressed or input file is depleted
  - o ***Display_get():*** dequeues a free video frame buffer from the V4L2 Display driver.
  - o ***fread():*** The next video frame is read from the file and copied into the dequeued video buffer.
  - o ***Display_put():*** Once the free video frame buffer has been written with valid video data, it is passed back to the V4L2 Display driver (enqueued) to be displayed.

17. **Build and install the application.**

18. **Check to make sure `video.raw` exists and has a file size larger than zero.**

    Navigate to **`/tmp`** in the EVM board's filesystem and list the contents of the directory. Use the "ls –lsa" flags to verify that `video.raw` exists and has a greater than zero file size. The application is hard coded (using a #define statement in `video_thread.c`) to read data from the file `/tmp/video.raw`.

    If you have powered off or reset the EVM since running the `lab07b_video_capture` application, the `video.raw` file will have been cleared from RAM memory. If so, go back and build/install `lab07b_video_capture` to create the `video.raw` file again.

19. **Execute the `./app_debug.xv5T` application. Press Ctrl-C to exit the application.**

    The application should play back the video from `/tmp/video.raw` along with your customized OSD banner (non-scrolling). The application does not actually modify the OSD, but the last OSD frame is likely still in the OMAP3530 Video Display Sub-system back end hardware and will therefore be displayed overlaid ontop of the video. If you reset the board, and execute lab07c_video_playback, you will see only played back video without the OSD overlay.

    Since there are only 10 video frames, you will have to look very closely to see movement, but the application pauses for 5 seconds after the 10 frames are displayed so that you will have time to see the final video frame displayed on the LCD.

Lab07d_video_loopthru

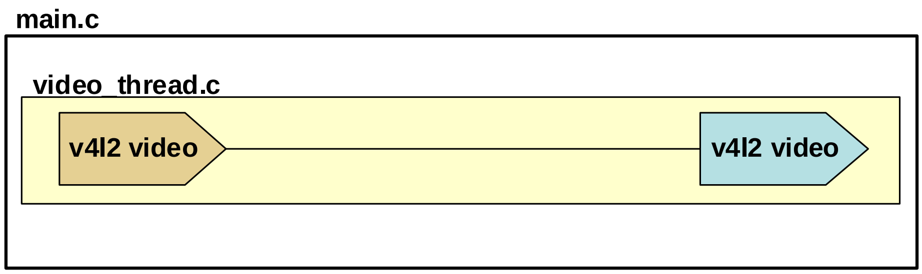
In this portion of the lab, you will combine the `lab07b_video_capture` and the `lab07c_video_playback` applications into a single video loop-thru application.

In part B, we recorded video from the v4L2 input and placed it into a file (video.raw) – this used an fwrite() command to write the video buffer to a file. In Part C, we did an fread() of the video.raw file and sent that video to V4L2 display driver.

We now have the input (capture) application (Part B) and the output (display) application (Part C) that you will now combine into a single application (Part D). We'll need to get rid of the "file reads/writes" and replace them with a memcpy operation to copy data from Capture driver buffers to Display driver buffers.

If using a memcpy to transfer the data between Capture and Display drivers seems like unnecessary overhead, you are correct! In lab07e, we will modify this application to pass the data via pointers. We are doing both labs for two reasons. Firstly, the pointer passing method is not as simple as may first seem, as will be explored in lab07e. Secondly, when we reach lab09 and add codecs to process our video data, the structure of lab07d will actually be more appropriate.

lab07d_video_loopthru



- **Goal: Combine the recorder (lab07b) and playback (lab07c) into a video loopthru application.**
- Hey – YOU get to do this yourself (no more inspection stuff…).
  1. Answer a few questions about the big picture (covered in the next few slides…).
  2. Copy files from lab07c (playback) to lab07d (loopthru).
  3. Add video *input* files from lab07b (record) to lab07d (loopthru).
  4. Make code modifications to stitch the record to the playback (covered in the next few slides…).
  5. Build, run. Result: video is captured (v4L2) and then displayed (FBdev) with your OSD.

Before we start copying, cutting, and pasting files and code, let's think about what must be done to get the loopthru lab to work.

- In `Lab07b_video_capture`, we used fwrite() to PUT (write) the video data to the `video.raw` file. What two functions were used to GET (read) the video data from v4L2 driver and return the video buffer back to the driver once the application has recorded the data?

```
GET video data:   1.  ________________________________________
                  2.  _ ______________________________________

PUT video data:   1. fwrite() the video frame ________________
```

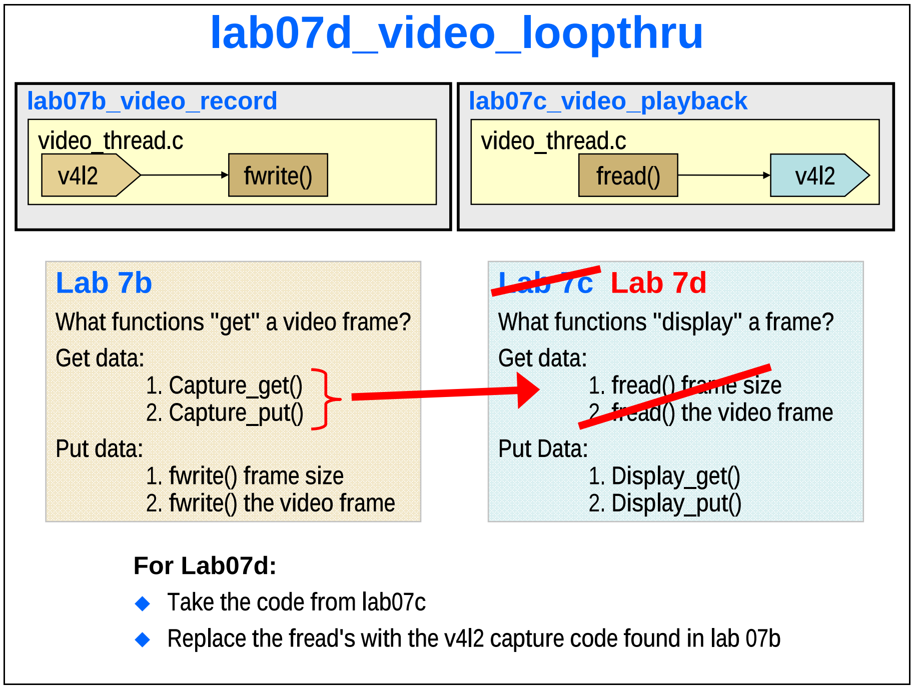
- Similarly, in `Lab07c_video_playback`, we used the functions listed below to PUT (write) the data to the FBdev driver. What function is used to GET (read) the video data?

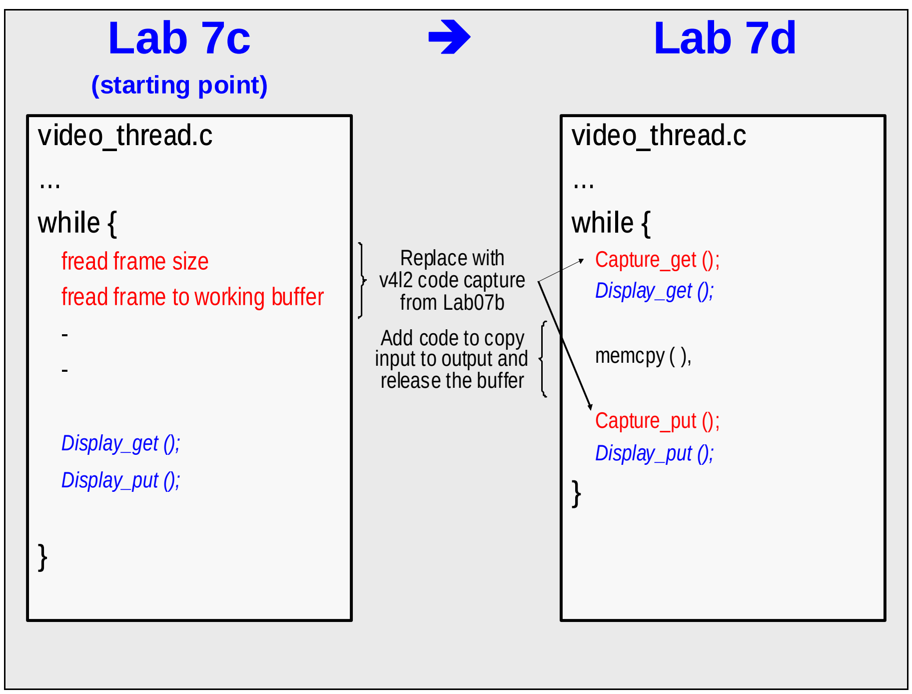
```
GET video data:   1.  ________________________________________

PUT video data:   1. Display_get() to get an empty video buff
                  2. Display_put() to display video data ____
```

In this lab exercise, we will start with the `Lab07c_video_playback` files, then edit them to create the loopthru code. Based on this, generally what functions should be required for our while() loop in the `Lab07d_video_loopthru`?

```
Get video data:   1.  ________________________________________

                  2.  ________________________________________

Copy video data:  1.  memcpy to copy from input to output ___

Put video data:   1.  ________________________________________

                  2.  ________________________________________
```

To summarize, the following lab procedure will take the _capture and _playback files and combine them into a loopthru example.





Note: For those advanced students who would like a challenge, see if you can accomplish this lab without referring to the procedure below. If you finish within 15 minutes with no help, you may ask the instructor(s) for a free 4Gen iPod touch 64G.

Lab07d Procedure

20. As a starting point, begin by copying the `lab07c_video_playback` application into the `lab07d_video_loopthru` folder.

```
(ubuntu) # cd /home/user/labs
(ubuntu) # cp –Rf lab07c_video_playback/* lab07d_video_loopthru
(ubuntu) # cd lab07d_video_loopthru/app
(ubuntu) # make clean
```

21. Reference the header file you just copied.

Open `lab07d_video_loopthru/app/video_thread.c` for editing along with `lab07b_video_capture/app/video_thread.c`

To open both files, you can either open a second terminal, invoke gedit with a trailing ampersand (&) to keep control of a single terminal, or supply both file names when you invoke your editor. You can also load the second file using the file drop-down menu.

22. Remove the INPUTFILE definition (lab07d).

Since our loopthru app won't need to read video from a file any longer, we don't need this definition any longer. In `video_thread.c`:

- Remove the #define constant declaration for INPUTFILE.

```
//* Input file *
//#define INPUTFILE "/tmp/video.raw"
```

23. Define that we want three capture buffers (and two display buffers).

Similar to that for display (which should already be defined from the playback application), we need to indicate to our thread the number of capture buffers used in the input driver using the NUM_CAP_BUFS constant.

```
//* Double-buffered display, triple-buffered capture *
#define NUM_DISPLAY_BUFS  2
#define NUM_CAPTURE_BUFS  3
```

Note: The display buffers are rotated using the hardware Video Rotation Framebuffer (VRFB) which requires a large VRFB buffer allocation (2048x640 pixels) due to the fact that it rotates a fixed 2048x2048 dataset. For this reason, it is recommended that two display buffers be used. (In the current configuration of the CMEM module, there is only enough pre-allocated memory to support two such buffers, so the curious student who modifies the NUM_DISP_BUFS to 3 will have a buffer allocation failure.)

24. Change `#define` constants so that our debug (i.e. printf) comments make sense.

This needs to be done in two places – where it's *defined* and *used*.

Change the INPUTFILEOPENED constant to CAPTUREDEVICEINITIALIZED.

First, we need to edit the initMask which originally followed the fopen() we deleted. It should now look like:

```
/* Record that capture device was opened in initialization bitmask */
initMask |= CAPTUREDEVICEINITIALIZED;
```

Then, go back up to the declarations section for *video_thread_fxn* and locate and change the appropriate #define statement. It should now look like:

```
/* The levels of initialization for initMask */
#define OSDSETUPCOMPLETE          0x1
#define DISPLAYDEVICEINITIALIZED  0x2
#define CAPTUREDEVICEINITIALIZED  0x4
```

25. Delete the code to open our input file.

Since we now need to read data directly from the video capture port – i.e. v4L2 via DMAI, we need to delete the code to open our input file.

In an earlier step we deleted the definitions for an INPUTFILE constant. Now, we need to delete its use.

Still within `lab07d_video_loopthru/video_thread.c`, locate the fopen() function that opens INPUTFILE for reading, then delete the entire *if* statement which contains it.

The statements to remove are:

```
if( (inputFile = fopen( INPUTFILE, "r" ) ) == NULL) {
    ERR( "Failed to open input file %s\n", INPUTFILE );
    status = VIDEO_THREAD_FAILURE;
    goto cleanup;
}

DBG( "Opened file %s with FILE pointer %p\n", INPUTFILE, inputFile );
```

26. Delete the code to read the input video standard from the input file

You should remove four lines, the actual fread call into the dAttrs.videoStd field, followed by the following two lines of error handling and the fourth line closing brace.

27. Delete the code to read the capture video size from the input file

You should remove four lines, the actual fread call into the captureSize field, followed by the following two lines of error handling and the fourth line closing brace.

28. Delete the line which sets the bufSize to captureSize as well as the printf of captureSize

bufSize variable will be determined in the capture code you are about to replace the above code with. You can printf the bufSize instead of removing this line if you like.

29. Replace the section that you removed in the previous 4 steps with the section from `lab07b_video_capture/app/video_thread.c` that initializes the capture device.

Be sure that this coded is added to **`lab07d/video_loopthru`** above the section of code initializing the display driver as the initialization of the video display driver is dependent upon the captureSize variable determined by the capture driver.

Note: we will add the necessary variable declarations in a later step.

The code that is necessary for the input capture setup is described below. You can either look at lab07b and figure out which code you need to copy or move down to the "spoiler alert" section below for more direct hints about what to do here.

The code you need to copy/paste from lab07b does the following:

- Detect the video standard on the video capture port (NTSC versus PAL)
- Calculate the video buffer size according to the input video standard
- Calculate the dimensions of the video buffer based on the input video standard and the (assumed) color space YUV (in UYUV format) and store in gfxAttrs structure.
- Create a table of buffers for use by the capture driver based on the size and attributes previously calculated.
- Create a reference to the input capture driver and store in hCapture variable

Spoiler Alert! (On following comments) You should attempt to determine which portion of code from `lab07b_video_capture` is necessary based on the information above, but as a double check, you will need everything from `lab07b_video_capture/app/video_thread.c` that starts at the comment banner:

```
/* Detect which video input is connected on the component input */
```

until you reach (but not including) the banner:

```
// Open the output file
```

30. Add the video capture declarations to the *video_thread_fxn()*.

Once again, since we copied the files from the *playback* directory, the display declarations should already be setup. Now, we just need to go back and add the capture declarations from the `lab07b_video_capture` directory.

We suggest to cut/paste these variables from `lab07b_video_capture/video_thread.c`.

We need:

- A video capture driver attributes structure, which will be named “cAttrs” and initialized to the DMAI video capture default values for the OMAP3530
- A DMAI handle to store our reference to the Capture driver once it is opened, which will be named “hCapture.”
- A video standard type enumerated variable named videoStd
- A DMAI Buffer table handle named “hBufTabCapture.”
- A buffer handle to refer to buffers interchanged with the capture driver, which will be named “cBuf.”

31. Configure the video display driver video standard to match that of the capture driver

Locate the code which sets the numBufs and rotation for the display attrs:

```
dAttrs.numBufs = NUM_DISPLAY_BUFS;

dAttrs.rotation = 90;
```

This code should remain unchanged. However, you want to add another line of code which will modify the videoStd field of the display attributes as:

```
dAttrs.videoStd = Capture_getVideoStd(hCapture);
```

32. In the video thread while loop, replace the video input functions — from *fread()* to the v4L2 capture driver.

. First, with all the cutting/pasting going on, make sure you are editing the correct file: `lab07d_video_loopthru/video_thread.c`.

Within the while loop of *video_thread_fxn*, we're going to replace the fread() statement with the code needed to capture the frame from the v4l2 device. Replace:

```
// Read raw video data from inputFile
if( fread( Buffer_getUserPtr(dBuf), 1, captureSize, inputFile ) <
captureSize )
    break;
```

with the code required to read from the v4l2 device. Again, it's probably easiest to cut/paste this from `lab07b_video_capture` – you will need two function calls – a _get call to get a handle to the next buffer on the capture driver queue and a _put call to place it back on the Capture driver queue after you are finished with it.

Hint: when finished with this editing, you should have the following *pseudo code*:

```
_get        //capture
_get        //display

**** to be filled with memcpy ****

_put        //capture
_put        //display
```

33. Insert a *memcpy* command to copy the video frame from your capture buffer to the display buffer.

(Later in the workshop, we will replace this with a codec/algorithm process call.)

What three arguments should we use for *memcpy*():

Hint: you will need to use the `Buffer_getUserPtr()` and `Buffer_getSize()` DMAI Buffer methods to determine this information from the Buffer objects pointed to by the cBuf and dBuf handles.

Both of these functions take a single argument, which is a Buffer handle, i.e.

```
void *myPtr;

myPtr = Buffer_getUserPtr(cBuf);
```

The previous statement would store the userspace pointer (i.e. virtual address) of the buffer referred to by the cBuf Buffer object handle into the myPtr variable.

```
int mySize;

mySize = Buffer_getSize(cBuf);
```

The statements above would store the size of the buffer referred to by the cBuf Buffer object handle into the mySize variable.

```
Destination:  ________________________________

Source:       ________________________________

Length:       ________________________________
```

memcpy has the following function prototype:

```
memcpy(void *destination, void *source, int length);
```

Note: Step 38 provides the "reality check" for this step -- what the interior of the `video_thread_fxn()` while loop should look like. You may either double-check your work now or proceed to step 37 and only use the double-check if the application does not perform as expected.

34. Remove the `sleep(5)` call after the while loop.

This function pauses five seconds, which was important for our playback application due to the small number of frames played back but is no longer necessary.

35. Cleanup the video input … rather than fclosing the input file.

Finally, replace the section in the cleanup that closes the raw <u>video input file</u> with the corresponding cleanup code from **`lab07b_video_capture`** that cleans up the capture driver.

36. Save and close `video_thread.c` from `lab07d_video_loopthru`.

Note: you should just close `video_thread.c` from **`lab07b_video_capture`**, because you should not have made any changes to this file.

37. Build and install the application…then run it.

Oh, and make sure your video source is still playing.

38. Reality check.

The interior of the `video_thread_fxn()` while loop should look as follows:

```
while ( !env->quit )
{
    Capture_get( hCapture, &cBuf );
    Display_get( hDisplay, &dBuf );
    memcpy( Buffer_getUserPtr(dBuf ),
            Buffer_getUserPtr( cBuf ),
            Buffer_getSize( dBuf ) );
    Capture_put( hCapture, cBuf );
    Display_put( hDisplay, dBuf );
}
```

Note: The size of the capture and display buffers is the same (as guaranteed in the initialization section of the `video_thread_fxn()`), so the memcpy command could also use Buffer_getSize(cBuf) as its final argument. More robust code could test and determine the lesser of the two sizes and copy only that much.

Lab07e_video_efficient

39. As a starting point, begin by copying the `lab07d_video_loopthru` application into the `lab07e_video_efficient` folder.

```
(ubuntu) # cd /home/user/labs
(ubuntu) # cp –Rf lab07d_video_loopthru/* lab07e_video_efficient
(ubuntu) # cd lab07e_video_efficient/app
(ubuntu) # make clean
```

40. Open `lab07e_video_efficient/app/video_thread.c` in the editor of your choice.

41. Define a combined capture and display number of buffers constant.

In order to use pointer passing between the Capture and Display drivers, we will need to initialize the drivers during Capture_create and Display_create with the same (shared) buffer pool.

It is a good idea to comment out or delete the *#define* statements for NUM_CAPTURE_BUFFERS and NUM_DISPLAY_BUFFERS so that the compiler will warn of any places where you might forget to replace them. After these have been removed, define a new constant NUM_CAPTURE_DISPLAY_BUFFERS and set the size to 3.

42. Declare a combined *hBufTabCaptureDisplay* buffer table handle.

Again, it is a good idea to remove the hBufTabCapture and hBufTabDisplay. You can use these previous buffer table handle declarations as a format for declaring hBufTabCaptureDisplay.

43. **Modify the *BufTab_create* for the *hBufTabCapture* to store the buffer table instead in *hBufTabCaptureDisplay*.**

    Don't forget to use your new constant NUM_CAPTURE_DISPLAY_BUFFERS when creating this buffer table.

44. **Modify *cAttrs.numBufs* to use the appropriate constant.**

    This attribute tells the Capture driver how many buffers to use from the provided buffer table during *Capture_create()*.

45. **Modify *Capture_create()* to use the new buffer table handle.**

    The V4L2 driver assigns an index to each buffer that it circulates. It is important that every buffer which will be passed to the Capture driver via your application is registered with the driver at initialization so that these indices may be mapped.

46. **After *Capture_create()*, reclaim all buffers and mark unused.**

    When the *Capture_create()* call is made with *hBufTabCaptureDisplay*, each buffer in this buffer table is not only registered with the capture driver, but queued onto the driver's incoming queue. We need to dequeue all buffers and mark them as free so that they will be registered with the display driver during *Display_create()*.

    Add a declaration for

    ```
    int i;
    ```

    at the top of `video_thread_fxn()` and then set up a for loop which iterates NUM_CAPTURE_DISPLAY_BUFS times and uses *Capture_get()* to reclaim and buffer *BufTab_freeBuf* to mark the buffer unused.

    Note: the Display driver will always hold one buffer in the pool (for display), so it is simpler to do *Capture_create()* first, reclaim all buffers, and then do *Display_create()*.

47. **Remove the *BufTab_create()* call to create a second buffer table for *hBufTabDisplay*.**

    Since we are sharing *hBufTabCaptureDisplay* between the two drivers, there is no need to create the second buffer table.

48. **Modify *dAttrs.numBufs* to use the correct number of buffers.**

49. **Modify *Display_create()* to use the *hBufTabCaptureDisplay* buffer table.**

50. **Get a single buffer from the Display driver and put it on the Capture queue.**

    As with the Capture driver, after *Display_create()* is called, all buffers from the provided buffer table are queued into the driver's incoming queue. Because the Display driver must always hold a buffer for display, we generally want to have a single buffer on the Capture queue and two buffers on the Display queue. Thus, before entering the while loop, we need to *Display_get()* a buffer from the Display driver and *Capture_put()* it onto the Capture queue.

    Hint: if you need help with the *Display_get()* and *Capture_put()* functions, you can reference these function calls within the while loop.

51. Within the *while loop*, remove the *memcpy* statement and instead use pointer passing to pass buffers between Capture and Display.

Since the *cBuf* and *dBuf* Buffer_Handle's both refer to buffers on the same *hBufTabCaptureDisplay*, you can simply pass *cBuf* to the Display via *Display_put()* and *dBuf* to the Capture driver via *Capture_put()*.

52. Save `video_thread.c`, then *make install* and test the application.

53. (Optional) Benchmark the application.

If you wish to benchmark this more efficient application you can execute it in the background using the "&" character.

```
# ./app_debug.xv5T &
```

Note, that Linux will display the process ID (PID) for `./app_debug.xv5T`.

Now run the "top" command:

```
# top
```

where you can then view the CPU % for the `./app_debug.xv5T` application. When ready, you can use Ctrl-C to exit the *top* application.

Since our `app_debug.xv5T` program is now running in the background, it won't receive the *kill* signal if we press Ctrl-C. (Notice just above, that Ctrl-C stopped top, but didn't affect our program.) In order to stop it, we need to send a signal directly to our program's process ID. We can do this with the *kill* command. (By the way, the SIGINT signal used below is the same signal that is normally generated by our Ctrl-C.)

```
# kill –s SIGINT [PID]
```

where [*PID*] is the process ID which was displayed when you launched `./app_debug.xv5T` in the background. If you can't find the PID in the terminal output, you can run the "*ps*" command to list all currently running processes and their PID's.

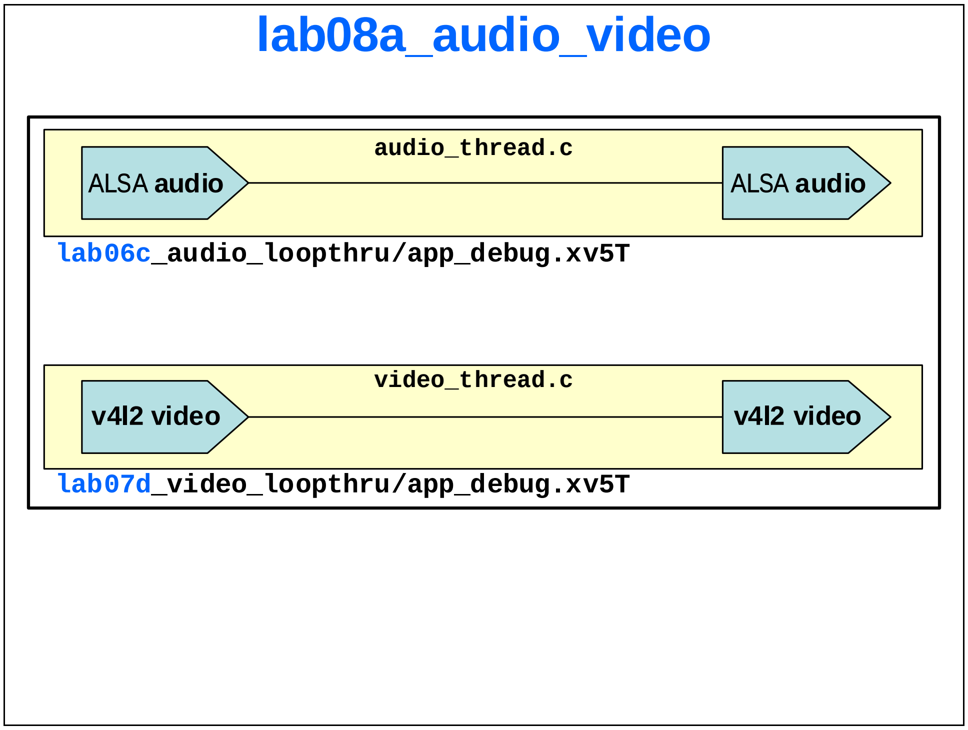
Lab 8 - Running Audio and Video

Introduction

Welcome to labs 8a and 8b. In these labs, you will combine the audio loopthru application from lab6c with the video loopthru application of lab7d into a single (multi-threaded) application that handles both audio and video.

Outline

Lab08a – Run Audio and Video in Separate Processes



Build Audio Executable

1. **Change to the `/home/user/workshop/lab06c_audio_loopthru/app` directory in the Ubuntu PC (solution to the previous audio loopthru lab).**

   Note, if you couldn't get lab06c working properly, copy from the solutions folder.

2. **Build and install the application using "`make debug install`".**

   This will build the debug version and install it to the DVEVM. Note, if you have problems building at this step, try cleaning, then building:

```
make clean
make debug install
```

3. **On the DVEVM board, use the Linux "mv" command to change the name of the `app_DEBUG.xv5T` application to `app_AUDIO.xv5T`.**

Build Video Executable

4. **Change to the `/home/user/workshop/lab07d_video_loopthru/app` directory.**

5. **Build and install the application.**

```
make debug install
```

Run Audio and Video in Separate Processes

6. On the DVEVM board, execute the `app_AUDIO.xv5T` application using the following command:

```
./app_AUDIO.xv5T &
```

Note, the trailing ampersand (&) in this command indicates that the application is to be run as a separate process. (In this case, our audio app will run in the terminal background, meaning that the terminal will remain open to new commands even while the application is executing.)

You may need to press the Enter key inside your TerraTerm terminal in order to get a new Linux command prompt.

7. On the DVEVM board, execute the `app_DEBUG.xv5T` application (the video loopthru application) using the following command:

```
./app_DEBUG.xv5T
```

You should now have both audio loopthru and video loopthru running concurrently on the board. They are running as concurrent, but separate, processes. In lab08b and lab08c we will use pthreads to run the audio and video loopthru in parallel threads within the same process or application.

8. Halt the video loopthru (running in the terminal foreground) by pressing Ctrl-C.

9. Use the following command to determine the process ID of the audio loopthru, which is running in the terminal background:

```
ps
```

Look for the `app_AUDIO.x470MV` to find its PID. To be more fancy, you could pipe the output of `ps` to the Linux `grep` command:

```
ps aux | grep "app_AUDIO.xv5T"
```

10. Halt the audio loopthru using the kill command and PID value from the last step.

```
kill –s SIGINT <app_AUDIO.xv5T process ID>
```

For example, if the process ID is 500, type:

```
kill –s SIGINT 500
```

Lab08a Question

. Which scheduling policy is being used by each of the audio and video program processes (i.e. how is the thread within each process being scheduled)? ___________________________

. ____

. ____

Lab08b_audio_video

In this lab, we will combine lab06c_audio_loopthru and lab07d_video_loopthru into a single, multi-threaded application. *(Note,if you were not able to get one of these labs to work, you can copy it from the appropriate solutions folder: `/home/user/solutions`)*

File Management

11. Change to the `/home/user/labs` directory.

```
cd ~/labs
```

Hint: **It is important to do the following two steps in this exact order.**
Otherwise, some of the following directions (i.e. editing main.c) will be incorrect!

12. List the lab08b_audio_video/app diretory

Two previously unused files have been provided for you:

```
# ls lab08b_audio_video/app

thread.c

thread.h
```

13. Examine thread.c in gedit or similar editor

This file contains one function, launch_pthread() which will takes five parameters:

- ***pthread_t *hThread_byref:*** this is a pointer to a handle (i.e. a handle passed by reference) which is used as a return value. The memory location pointed to by hThread_byref will be updated with a pthread handle.
- ***int type:*** integer specifying that the created thread will either be REALTIME or TIMESLICE as per #define in thread.h
- ***int priority:*** The priority assigned to the thread. (is only used for thread type = REALTIME)
- ***void *(*thread_fxn)(void *env):*** a pointer to the function that is the entry point for the created thread. This function takes a single pointer as an argument (although the pointer may be a pointer to a structure, effectively allowing multiple arguments.)
- ***void *env:*** the pointer which will be passed as the argument to `thread_fxn()` as per the above

Examination of the launch_pthread() function shows the thread creation procedure which was reviewed in the lecture portion of module 8.

14. **Copy the full contents of `lab07a_osd_setup` into `lab08b_audio_video`.**

    ```
    cp –R –f lab07a_osd_setup /* lab08b_audio_video
    ```

    or you can use the file browser within Ubuntu.

15. **Copy the full contents of `lab06c_audio_loopthru` into `lab08b_audio_video`.**

    ```
    cp –R –f lab06c_audio_loopthru/* lab08b_audio_video
    ```

    or you can use the file browser within Ubuntu.

16. **Copy the full contents of `lab07d_video_loopthru` into `lab08b_audio_video`.**

    ```
    cp –R –f lab07d_video_loopthru/* lab08b_audio_video
    ```

    Don't worry about overwriting any files.

Edit main.c

17. Open `lab08b_audio_video/app/main.c` in a text editor.

18. Fill in the missing `.h` files, as well as the missing `_env` variables for the audio and osd threads to `main.c`.

Your `main.c` file should contain the following:

```
video_thread.h
video_env         (which is the video_thread_env variable)
```

You need to add the following to `main.c`. (Refer to lab06c for this code.)

```
audio_thread.h
audio_env         (which is the audio_thread_env variable)
```

as well as (Refer to lab07a for this code.)

```
osd_thread.h
osd_env           (which is the audio_thread_env variable)
```

19. Make sure that video, audio and osd *while* loops exit when Ctrl-C is pressed.

Recall that the signal_handler function is run whenever Ctrl-C is pressed. This signal handler sets the quit field in both of these global structures to true, signaling to the thread that it should proceed to its cleanup phase and then exit.

How do the threads know where to look for these variables? These environment structures are passed as the argument to the thread. Within the thread function, the main while loop tests on the appropriate quit variable. When the quit variable becomes true, execution drops out of the while loop and into the final (cleanup) phase of the function.

Currently the *signal_handler()* function sets *video_env.quit* to one (true). We need to add similar statements for the *audio_env.quit* and *osd_env.quit* to the signal handler below.

```
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting..\n");

    _______________________________

    video_env.quit = 1;
}
```

20. To make debugging easier, put a one-second delay in between *.quit=1 calls.

Since we're exiting three pthreads back-to-back, you might find that their debug messages become interleaved – which can make debugging more difficult. To this end, when building with our "debug" profile, we could delay the start of the second *quit* by using a Linux time function.

Insert the following code between each **.quit=1* call to cause Linux to sleep for one second. This should make debugging easier.

```
#ifdef _DEBUG_
    sleep(1);
#endif
```

Don't forget to include the proper header file for the *sleep()* function: `unistd.h`

21. Include the `<pthread.h>` header file that prototypes the launch_pthread() function.

This is the header file for the code you examined in step 13.

22. Declare three pthread handles and three return pointers needed to manage our new threads (audio, video, osd).

At the top of main(), add three pthread handles (of type *pthread_t*) named *audioThread, osdThread* and *videoThread*. Handles are used to refer to instantiated objects; the handle value will be set during *pthread_create()* in step 23, then used again later to refer to that pthread instance.

Also, we want to add three void pointers: one named *audioThreadReturn*, one named *osdThreadReturn* and one named *videoThreadReturn*. The return pointers will be used when "joining" (i.e. exiting) a thread in step 26; they will allow you to interrogate the status after an exit.

It should end up looking like:

```
pthread_t  audioThread, osdThread, videoThread;
void       *audioThreadReturn, *osdThreadReturn, *videoThreadReturn;
```

23. Replace the direct call of `video_thread_fxn()` **with a call of `launch_pthread()` to create a new thread that has `video_thread_fxn()` as its entry point.**

Currently `main.c` calls *video_thread_fxn*. Replace this direct function call with a call to launch_pthread()

Replace the following:

```
/* Call video thread function */
videoThreadReturn = video_thread_fxn((void *) &video_env);
```

With this starting code … you have to fill in a few details:

We've given you hints, but you need to figure out the actual arguments…

```
/* Create a thread for video loopthru */
if(launch_pthread(&thread, (type), (priority), video_function, argument)
!= thread_SUCCESS){
      ERR("pthread create failed for video thread\n");
      status = EXIT_FAILURE;
      goto cleanup;
   }
   initMask |= VIDEOTHREADCREATED;
```

Don't forget to add #defines for audio & video thread created masks.

(type) – either REALTIME or TIMESLICE as per thread.h

As per the lecture, we want to set this field to TIMESLICE for video_thread_fxn()

`(priority) – this value will be ignored for TIMESLICE type, so you can set it to zero`

24. Add launch_pthread() calls to launch the audio_thread_fxn() and osd_thread_fxn() entry points as threads with the following characteristics:

audio_thread_fxn():	type: REALTIME
	priority: 99
osd_thread_fxn():	type: TIMESLICE
	priority: 0 (unused)

At this point, you should have 3 sets of launch_pthread calls (one for video, audio, and osd).

Be sure to record successful launching of the audio and osd threads in the intiMask.

25. To make debugging easier, put a one-second delay in between *pthread_create()* calls.

Since we're creating two pthread's back-to-back, you might find that their debug messages could become interleaved – which can make debugging more difficult. To this end, when building with our "debug" profile, we could delay the start of the second *pthread_create()* by using a Linux time function.

Insert the following code between your two *pthread_create()* calls to cause Linux to sleep for one second. This should make debugging easier.

```
#ifdef _DEBUG_
    sleep(1);
#endif
```

26. Add "cleanup" section using *pthread_join()* for both audio and video threads.

First, let's create a "cleanup" section in our `main.c` file.

After the audio, osd and video threads have been created, use *pthread_join* on all three threads to pause execution of the main thread until all threads have exited.

The prototype for pthread_join is:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The first parameter is the handle to the thread to join to (the variable we created in step 22, then filled-in with pthread_create() in step 23).

We use audioThreadReturn, osdThreadReturn and videoThreadReturn pointers (by reference) to store the return status (pass/fail) from the join function. Since we want to return a value via this argument, we want to pass the (void pointer) argument by reference. To avoid getting an incompatible pointer type warning, we want to recast this argument – since this recasting can be a bit tricky for some of us, rather than have you figure it out by trial-and-error, here are the values to use for the second argument of each join function:

```
(void **) &videoThreadReturn
(void **) &osdThreadReturn
(void **) &audioThreadReturn
```

So, as an example, to join the video thread, you can write the following after "cleanup:" :

```
Pthread_join(videoThread, (void **) &videoThreadReturn);
```

27. Ensure all `initMask` #defines are completed for video, audio and osd.

28. Save and close `main.c`.

29. Modify your `makefile_profile.mak` to copy (i.e. install) the `ti_rgb24_640x80.bmp` file from the osdfiles directory to the execution directory.

The way this install rule is written it will copy all of the dependencies of the rule into the execution directory. Knowing we would have an OSD file eventually, we added an OSD variable to the install string. As long as INSTALL_OSD_IMAGE = “”, nothing was copied.

We can change this by adding the file we want copied to the Application Information section of our makefile. That is, set the INSTALL_OSD_IMAGE variable to the name of your OSD image file. (Make sure it starts with “`../osdfiles/`” so gMake can find it.)

```
PROGNAME          := app
CONFIG            := app_cfg
INSTALL_OSD_IMAGE := ../osdfiles/ti_rgb24_640x80.bmp
INSTALL_SERVER    :=
  ...
.PHONY  : install
install : $(PROGNAME)_$(PROFILE).xv5T $(INSTALL_OSD_IMAGE) $(INSTALL_SERVER)
     (install rule continues…)
```

Alternatively, we could have just had you add the OSD filename to the install rule dependencies like this.

~~`install : $(PROGNAME)_$(PROFILE).xv5T ../osdfiles/ti_rgb24_640x80.bmp`~~

We chose not to do this so that we could try and keep all the application specific info grouped together towards the top of the makefile.

30. Build and install the application using gMake:

```
make debug install
```

31. Execute the `./app_DEBUG.xv5T` application on the development board.

You should have simultaneous audio and video playing through the board, with a scrolling OSD.

32. Press Ctrl-C to exit the application.

Lab08b Question

. What scheduling policy is being used by each of the audio, video and osd threads? _______

. ____

. ____

Chapter 8 Appendix

Sidebar - Looking at the pthread arguments in detail:

The detailed function prototype for pthread_create is:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                void *(*start routine)(void *), void *arg);
```

***thread**: After pthread_create() runs, the first argument becomes our handle to the newly created thread instance. We'll use it every time we want to do something with/to this specific thread instance.

The handle is of type *pthread_t* (i.e. **pthread_t**ype).

It is passed by reference (hence *thread in the above prototype), which allows the pthread_create() to return the value for our newly created thread.

A final note (to those of us who are a bit rusty on our C syntax), if the pthread_create() function is going to use this argument as a pointer, then we need to pass it the address (&hint, hint) of our pthread_t variable.

***attr**: The second argument is a pointer to a thread attributes structure. Hence, it uses a variable of type **pthread_attr_t**ype.

In the next lab we will modify the thread attributes, but for now will use default thread attributes by passing a NULL pointer.

start_routine:

The third argument is both the easy and hard to understand. Let's focus on the easy part here. Simply, you just need to specify the name of the function to be run once the thread is created. As a hint, ask yourself this question, what function call are we replacing in main() with pthread_create()? That is the function we need to enter here as the 3rd argument.

(See the sidebar at the end of this step for a discussion of this arguments "structural complexity".)

***arg:** The final argument to pthread_create is a void *argument. When start_routine() is run, upon creating our pthread, this is the argument that will be passed as the start_routine's one-and-only argument.

In our case we want to pass *video_env* to *video_thread_fxn* and *audio_env* to *audio_thread_fxn*. We can do this by passing both structures by reference and recasting to a void pointer type, i.e.:

```
(void *) &audio_env
(void *) &video_env
```

Sidebar to the Sidebar – The devilish details of the pthread_create() 3rd arg

The third argument to the pthread_create() function specifies the start_function. That is, the function automatically run after creating the new thread.

As an example, the argument might look like:

```
pthread_create(&myThread, NULL, myFunction,(void *) &myArg)
```

Looking at the *official* definition for the pthread_create() function, we find the third argument looks like:

```
void *(*start_routine)(void *)
```

All this is really saying is that this argument is just a pointer to a function whose prototype is:

```
void *start_routine(void *arg);
```

Note, our example's prototype would be:
void *myFunction(void *arg);

This prototype means that it is a function that takes a void pointer as its single argument and returns a void pointer.

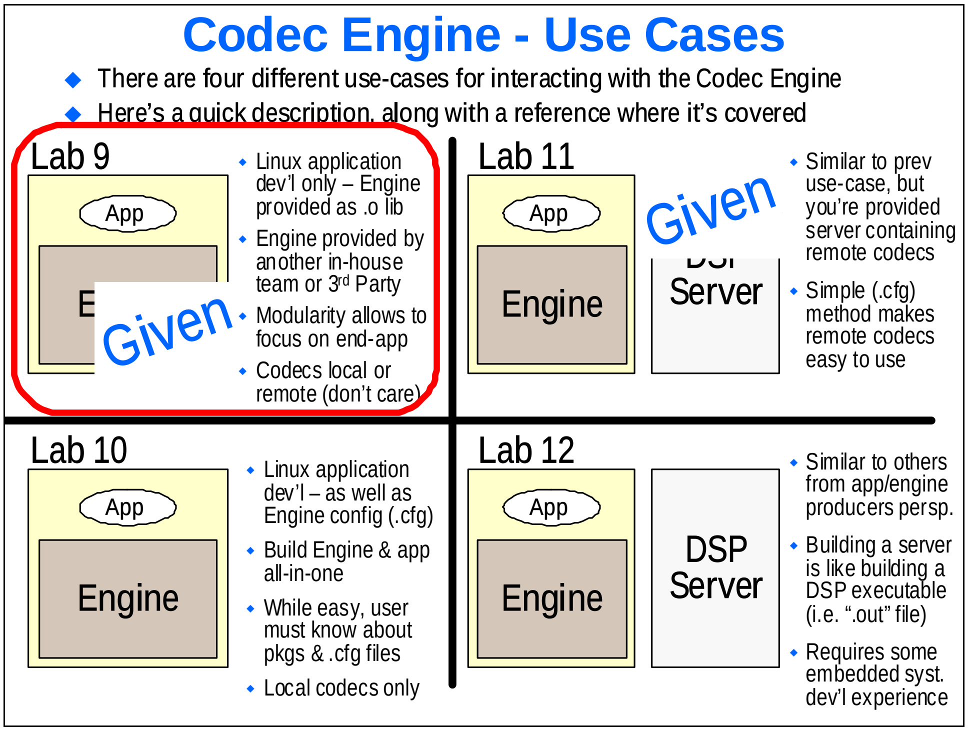
Why does the pthread definition use the extra complication of void pointers for both the argument and return values? Because this flexibility allows you to create a function that meets your needs. You can define any structure (or scalar) to be passed and returned to your start function.

In other words, the thread function is defined with void pointers for both its argument and return because this allows you to create any structure you wish for each of them. This allows you to populate the argument structure with as many arguments as you want – ditto for the return structure – and pass pointers to these structures.

Fortunately (and not by accident…) the *video_thread_fxn* and *audio_thread_fxn* that we have been using happen to both use void pointers as their argument and return values.

Lab 9 - Using a Given Engine

Lab 9 – 12 Summary



Lab Outline

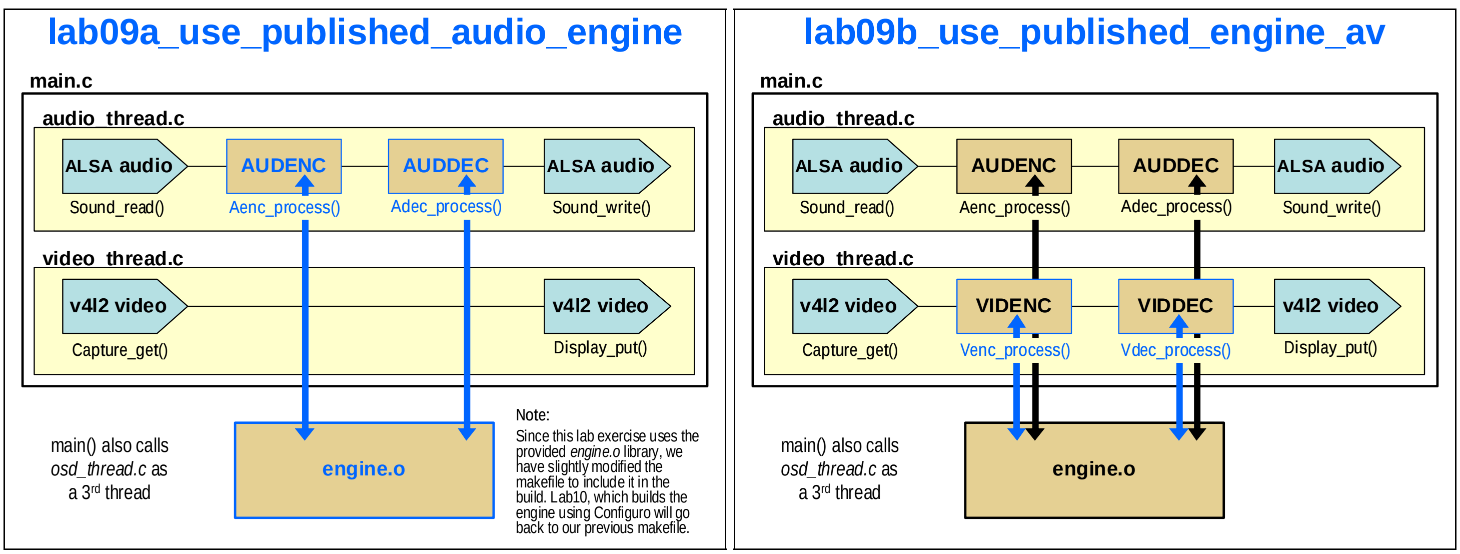
Lab 9 Introduction

In this lab exercise, you will extend the Lab8b audio/video example to add an audio and video encoders and decoders to your audio and video streams.

- Specifically, in Lab9a we provide the code to add an audio encoder & decoder to your audio loopthru thread from Lab8b.
- Then, in Lab9b, you will add video encoder & decoder processing to the video thread.

This lab makes use of the dummy (i.e. pass-thru) audio and video codecs that come with the Codec Engine examples, running locally on the ARM processor. These dummy codecs are a great way to build-up (i.e. model) a system prior to having your completed algorithms ready to integrate. In our case, they provide a simple first step towards integrating signal processing.

The project uses a pre-built (or "published") engine; essentially, the entire signal processing content has been wrapped up into a single library archive. On your end, though, in order to utilize the audio and video codecs from the published engine, you will need to add Engine and VISA functions to your application. (The next chapter discusses how to build an engine.)



Lab09a_use_published_engine

To reconfirm, in this lab you should hear audio and see video, but we are only "processing" the audio. That is, we will continue to use a ***memcpy()*** to pass-thru the raw video data. On the other hand, the audio data will be passed to the audio encoder – then audio decoder – before being written to the audio output driver. (Lab09b gets us processing the video, too.)

Examine Provided Lab Files

1. Change to the `/home/user/labs/lab09a_use_published_audio_engine/engine` directory and list the contents.

The `engine` directory contains two object libraries named: `engine_debug.ov5T` and `engine_release.ov5T`. These files contain all object code needed for our *given* engine, to be used by your application. Hence these are published as *pre-built* engines. (In lab10 we will examine how to build and modify an engine.)

In addition, we have included the `engine_cfg.cfg` file used to create these pre-built engines. Although this file will not be directly used by the *lab09a_use_published_engine* application, it's good practice to provide the original config file along with the engine, as a reference.

2. Change to the `app` directory and examine the files:

audio_thread.c

`audio_thread.c` should look very similar to the file you developed in Lab8. The following additions have been made to support audio processing using the Codec Engine:

- ***Added DMAI header files:*** Two additional DMAI header files have been #included into the file, `<ti/sdo/dmai/ce/Adec.h>` and `<ti/sdo/dmai/ce/Aenc.h>`
- ***A number of variables have been declared:*** Handles for the engine, audio encoder and decoder (*engineHandle*, *encoderHandle* and *decoderHandle*) as well as configuration parameters and dynamic parameters for the encoder and decoder (*aeParams*, *adParams*, *aeDynParams*, *adDynParams*). Note that the parameters structures have all been initialized to the DMAI default values in these declarations.
- ***Encoder/decoder creation:*** During the initialization stage of the thread, there is an *Engine_open()* call, followed by *Aenc_create()* and *Adec_create()* function calls.
- ***Encoded buffer create:*** A new buffer has been created to handle the encoded data coming from *Aenc_process*() before it is passed to the decoder. A reference to the buffer is stored in the *hBufEnc* handle.
- ***Within the while loop:*** *Aenc_process()* and *Adec_process()* have been called in between the *Sound_read()* and *Sound_write()* calls; these encode the audio putting it into the *hBufEnc* buffer, then decodes it again before output. Note the workaround: *Buffer_setNumBytesUsed()* is called after both *Aenc_process()* and *Adec_process()*; this is required because the CE copy codec examples don't set the size of their respective output buffers (because they're always the same size as the input buffers.) A production-quality codec should not require this workaround.
- ***Cleanup:*** After the while loop exits, *Adec_delete()*, *Aenc_delete()* and *Engine_close()* are called to free codec & Codec Engine resources. Also, *Buffer_delete()* was added to free the intermediate buf we created (ref'd by *hBufEnc*).

Examine main.c

Remember that *CERuntime_init()* must be called before any other Codec Engine functions. You might also note that the *audio_env* structure has been modified to include the string value for our *engineName*, which allows us to pass the name of our Engine from *main()* to the audio thread. (Each thread has to call *Engine_open* separately to obtain a unique handle to the Engine).

Build, Install, Run Application

3. Build and install the application using gMake (debug profile).

The **`video_thread.c`** file we have provided for you has a number of **`#define`** statements and variable declarations that have been added for convenience as outlined in lab0**9b**. As a result, when you build, you will get a number of warnings for unused variables:

```
video_thread.c: In function 'video_thread_fxn':
video_thread.c:95: warning: unused variable 'dDynParams'
video_thread.c:94: warning: unused variable 'eDynParams'
video_thread.c:93: warning: unused variable 'dParams'
video_thread.c:92: warning: unused variable 'eParams'
video_thread.c:91: warning: unused variable 'decoderHandle'
video_thread.c:90: warning: unused variable 'encoderHandle'
video_thread.c:89: warning: unused variable 'engineHandle'
video_thread.c:86: warning: unused variable 'encBufSize'
video_thread.c:85: warning: unused variable 'encBufAttrs'
video_thread.c:84: warning: unused variable 'encBuf'
```

Don't worry about these warnings – we'll use all of these variables in part B.

4. Insert the CMEM and DSPLINK drivers into the kernel.

If you have called the `./loadmodules.sh` script already you do not need to call it again, but if you have reset your board since the last time this was called, you will need to load those modules again.

Remember, in order to use a driver it must be installed into the kernel. Starting with Lab 9, we begin using CMEM. (DSPLINK won't be used until Lab 11.)

The `loadmodules.sh` script dynamically loads the `cmemk.ko` and `dsplinkk.ko` kernel modules using the *modprobe* command (which calls *insmod*) and allocates the appropriate device nodes to support the drivers using *mknod*, as we discussed back in Chapter 6.

On the EVM board (i.e. Tera Term), run the script:

```
./loadmodules.sh
```

Note: If loadmodules is not in our EVM's /opt/workshop directory, you missed a step from an earlier lab. No need to worry – in Ubuntu, run the install script from Lab00.

---

5. Execute the app_debug.xv5T application.

After you've confirmed it works, press Ctrl-C to exit the application.

Lab09b_use_published_engine_av

Using `lab09a_use_published_engine` as a reference, add a video encoder and decoder into the audio thread of the application. After copying over the previous lab, you will need to modify `video_thread.c` to use:

– Venc_create()	– Vdec_create()
– Venc_process()	– Vdec_process()
– Venc_delete()	– Vdec_delete()

As a hint, you may want to use `audio_thread.c` as a reference, even cutting and pasting segments from this file as a starting point for your code.

File Management

6. Begin by copying the files of `lab09a_use_published_audio_engine` into `lab09b_use_published_engine_av`.

```
cd ~/labs
cp –R -f lab09a_use_published_audio_engine/* lab09b_use_published_engine_av
```

Examine/Modify video_thread.c

7. **Open `lab09b_use_published_engine_av/app/video_thread.c` for editing.**

8. **Verify the appropriate #includes and #defines are present, which are used to access to the video decoder and encoder.**

    Note: Since this step is little more than an exercise in typing, we have provided the header file includes and defines for you in the starter file so that you can get on to the interesting portion of the lab. Each step that is already done has been outlined below.

    The following shows the header file includes that are needed to get our new codecs working.

a

```
/* Codec Engine headers */
#include <xdc/std.h>
#include <ti/sdo/ce/Engine.h>
```

b

```
/* DMAI headers */
#include <ti/sdo/dmai/Dmai.h>
#include <ti/sdo/dmai/Capture.h>
#include <ti/sdo/dmai/Display.h>
#include <ti/sdo/dmai/Buffer.h>
#include <ti/sdo/dmai/BufferGfx.h>
#include <ti/sdo/dmai/Vdec.h>
#include <ti/sdo/dmai/Venc.h>


/* Application header files */
#include "debug.h"                // DBG and ERR macros
#include "video_thread.h"         // video thread definitions
```

c

```
/* Video encoder and decoder used */
#define VIDEO_ENCODER "video_encoder"
#define VIDEO_DECODER "video_decoder"
```

```
...
```

```
/* Numbers of video buffers needed */
#define NUM_CAPTURE_BUFS 3
```

d

```
/* Create a joint buffer table to be shared between decoder and display *
//#define NUM_DISPLAY_BUFS 2
//#define NUM_DECODER_BUFS 3
#define NUM_DECODER_DISPLAY_BUFS 4
```

```
...
```

e

```
/*  Intermediate buffer for encoded video */
Buffer_Handle    encBuf      = NULL;      // pointer to encoded buffer
Buffer_Attrs     encBufAttrs = Buffer_Attrs_DEFAULT;
int              encBufSize  = 0;

/*  Codec engine variables */
Engine_Handle        engineHandle  = NULL;    // handle to Engine
Venc_Handle          encoderHandle = NULL;    // handle to video encoder
Vdec_Handle          decoderHandle = NULL;    // handle to video decoder
VIDENC_Params        eParams       = Venc_Params_DEFAULT;
VIDDEC_Params        dParams       = Vdec_Params_DEFAULT;
VIDENC_DynamicParams eDynParams    = Venc_DynamicParams_DEFAULT;
VIDDEC_DynamicParams dDynParams    = Vdec_DynamicParams_DEFAULT;
```

Discussion:

If you are wondering *"How on earth would I know what header files to include and variable types to define???"*, don't forget about the DMAI documentation. Due to time limitations in the workshop, it isn't efficient to have you search through the docs and header files, finding every type definition and enumeration value that you need. But alas, it's all there in the API reference guide documentation:

`~/ti-dvsdk_omap3530-evm_4_00_00_17/dmai_2_05_00_18_ApiReference.html`

You can view this file using the web browser installed on this system.

```
# cd /home/user/ti-dvsdk_omap3530-evm_4_00_00_17/dmai_2_05_00_18/
# firefox file://$PWD/dmai_2_05_00_18_ApiReference.html
```

Select the *Venc* module and note that all of the type definitions, functions, and default parameter structures are listed.

You may also find it convenient to look directly inside of the `Sound.h` DMAI header file. To do this, open a new terminal. Navigate to:

`/home/user/ti-dvsdk_omap3530-evm_4_00_00_17/dmai_2_05_00_18/packages/ti/sdo/dmai/ce`

and open the `Venc.h` header file in the editor of your choice.

Below is an explanation of each item we have done for you:

a. First, we need to include the header files that reference the standard *XDC definitions*, as well as the *codec engine* definitions.

b. We will be utilizing two new DMAI library modules to access video encoder and decoders, namely the *Venc* and *Vdec* modules. In this section, the header files for those modules are added. (These DMAI modules are basically wrappers around the Codec Engine functions we discussed earlier in the workshop.)

c. In our video thread, rather than using the actual codec string names (as we defined in our `.cfg` file), we chose to abstract them via `#define` statements.

d. For increased efficiency, we've chosen to share buffers between the *decoder* and *display*. Thus we commented out the previous statements and defined a new value. (Note, we were able to reduce the required buffers from 5 down to 4.)

e. We need to add declarations for each variable we plan to use in the video thread. Normal practice would obviously be to write the new code you are adding, then go back and declare variables as you need them. To save you some typing, though, we've already added them.

9. Extend the #define'd bit values for the *initMask* to include the following:

```
/* The levels of initialization for initMask */
#define     OSDSETUPCOMPLETE              0x1
#define     DISPLAYDEVICEINITIALIZED      0x2
#define     CAPTUREDEVICEINITIALIZED      0x4
#define     VIDEOENCODERCREATED           0x8
#define     VIDEODECODERCREATED           0x10
#define     ENCODEDBUFFERALLOCATED        0x20
#define     ENGINEOPENED                  0x40
```

Once again, we will use these in our cleanup & error management code to detect which actions have (or have not) been completed successfully.

Modifying video_thread.c (cont'd)

Add Codec Engine & Codec Calls

10. **Open the *codec engine* and create instances of both the video *encoder* and *decoder*.**

    DMAI provides functions for setting up these entities. (In fact, we added header files already for referencing these functions in a previous step (8b)).

    In this step you need to add function calls to:

    a. Open the Engine

    b. Create instance of the video encoder

    c. Create instance of the video decoder

    The code provided below needs to be added to `video_thread.c`. As shown below, we've provided the code for (a) **opening the Engine** and (b) **creating an instance of the encoder** as a reference. We've left it up to you to write the third piece (decoder instance), which also must be added to `video_thread.c`.

    You have two choices at this point: (1) type it all manually; (2) copy the three parts from the *audio_thread_fxn()* in `audio_thread.c` and paste them into `video_thread.c` – modifying them to use "video" references instead of "audio". It's your choice….

video_thread.c

```
/* Open the codec engine */
/* Note: codec engine should be opened in each thread that uses it */

engineHandle = Engine_open( env->engineName, NULL, NULL );


if(engineHandle = NULL){
        ERR( "Engine setup failed in video_thread_fxn\n" );
        status = VIDEO_THREAD_FAILURE;
        goto cleanup;
}
initMask |= ENGINEOPENED;

/* Create an instance of the video encoder           */

encoderHandle = Venc_create( engineHandle, VIDEO_ENCODER, &eParams,  &eDynParams );


if(encoderHandle = NULL){
        ERR( "Video encoder create failed in video_thread_fxn\n" );
        status = VIDEO_THREAD_FAILURE;
        goto cleanup;
}
initMask |= VIDEOENCODERCREATED;
```

. What is the engine name we are using and where is it defined? ______________________

. ____

. Trace out how the engine name gets from where it's defined, to where it is used in

. the *Engine_open()* function call? ____

. ____

Modify Buffer Table and Open Display Driver

11. In the *declarations* section, change `hBufTabDisplay` to `hBufTabDecoderDisplay`.

Updated to indicate we will be using a shared buffer between the decoder and display. (Strictly speaking, we didn't need to change this. But it often helps to keep variable names close to their usage/meaning.)

12. Modify the `hBufTabDisplay` *BufTab_Create()* to create a buffer table in `hBufTabDecoderDisplay` and move the create call to just after *Vdec_create()*.

Certain video codecs, such as H.264, use the concept of B frames which allow backwards references (from the encoder standpoint) to frames which have yet to be decoded. The result is that these decoders may need to maintain a pool of video frames that are operated upon together during process.

While not all decoders require this (certainly not the "copy" decoder!), we will go ahead and add it for when we use a real video codec in Lab 11a.

Since we will no longer need the Display buffer table (as we're now sharing a buffer table between the display driver the decoder) the simplest thing for you to do is cut and paste the DecoderDisplay buffer table allocation code and modify as necessary.

Paste just before the code that opens the video display – Display_create():

```
hBufTabDecoderDisplay = BufTab_create( NUM_DECODER_DISPLAY_BUFS,
                                       bufSize,
                                       BufferGfx_getBufferAttrs(&gfxAttrs));
if ( hBufTabDecoderDisplay == NULL )
{
     printf( "Failed to allocate contiguous buffers\n" );
     goto cleanup;
}

/* Create the display driver instance */
...
```

13. Modify the *Display_create()* call to use `hBufTabDecoderDisplay`.

Note: You may recall that in lab07e, it was necessary to dequeue the buffers – which were initialized in the capture driver – before the *Display_create()* call. In the case when using Vdec, the buffer table is registered but not queued (and not marked as in use), so there is no need to go through this procedure before sharing the buffer table with the Display driver.

14. Initialized video decoder to use `hBufTabDecoderDisplay`.

`Vdec_setBufTab()` grants the table of video buffers to the video decoder for its use.

We want to do this after just after the decoder instance has been created. So, add this code right after the *Vdec_create()* call (and it's associated error checking if statement):

```
/* For Vdec, we'll reuse the Buffer table created for the display */
Vdec_setBufTab(decoderHandle, hBufTabDecoderDisplay);
```

Create an intermediate buffer (encBuf)

15. First, let's allocate a buffer to hold data between encoder and decoder process calls.

Here's how we'll create our intermediate buffer:

```
encBuf = Buffer_create( bufSize,  &encBufAttrs );
```

Before we can execute this call, though, we'll first need to figure out it's proper arguments:

- Before we can allocate the buffer, we need to figure out how large it should be.
  - For a "real" codec, we could check the codec's datasheet for the maximum buffer size – or, some codecs provide this as a *status* value that can be returned via their *_control()* function.
  - In any case, since we are using dummy codecs (shipped with the Codec Engine), this is not a question because the output ("encoded") buffer size will always be the same size (the same data!) as the input buffer. *(To put this another way, since this is a dummy copy codec, we can cheat and know that this is always the right size:* `encBufSize = bufSize;`)
  - Note, *bufSize* is the video buffer size as calculated earlier in the video thread based upon the result of the *Capture_detectVideoStd()* call.
- For cache efficiency (on the DSP), we chose to align our buffers to BUFSIZEALIGN (set to 128 bytes, the cache line size of the C64x+ DSP).
- To simplify setting the size of the buffer, we called a DMAI function to choose between our two possible sizes (*encBufSize*, *BUFSIZEALIGN*).
- As always, it's a good idea to add error-checking code to help with debugging.

Here's the code you need to add to your file:

Add just after the call to *Vdec_setBufTab()* and right before the *while()* loop.

```
/* Set buffer size for intermediate buffer (for encoded data)  */
/* as the size of a full frame                                  */
    encBufSize = bufSize;

/* Allocate intermediate buffer                                 */
/* Note, must use contiguous buffers if passed to DSP!          */
    encBufAttrs.memParams.align = BUFSIZEALIGN;
    encBufAttrs.memParams.type  = Memory_CONTIGPOOL;

    encBuf = Buffer_create( Dmai_roundUp( encBufSize, BUFSIZEALIGN ),
                            &encBufAttrs );

    if ( encBuf == NULL )
    {
        ERR( "Failed to alloc video buffer in video_thread_fxn.\n" );
        status = VIDEO_THREAD_FAILURE;
        goto cleanup;
    }

    initMask |= ENCODEDBUFFERALLOCATED;

    DBG( "Alloc'd intermediate video buffer of size %d\n", encBufSize);
```

Update video thread's while() loop - Replacing memcpy() with video codec processing

13. Replace the memcpy in the "while" loop with processing calls to video encoder and decoder.

Within the *while* loop (in `video_thread.c`), you should find the following code:

```
get() a buffer from video capture device
get() a buffer from video display device
memcpy() captured buffer into display buffer
put() the capture buffer back to capture driver for reuse
put() the display buffer back to display driver for reuse
```

You need to replace the *mempcy()* function call with two calls to encode/decode the audio. Also, in our solutions we chose to put back the buffers right after their use – therefore, our pseudo-code looks like:

```
get() a buffer from video capture device
Venc_process() captured buffer into encoded buffer (encBuf)
put() the capture buffer back to capture driver for reuse

get() a buffer from video display device
Vdec_process() encoded buffer into display buffer
put() the display buffer back to display driver for reuse
```

As a hint, here is the prototype for *Venc_process()*, the DMAI video encoder process function.

```
int Venc_process(Venc_Handle encoderHandle, Buffer_Handle inputBuffer,
                                            Buffer_handle outputBuffer);
```

```
Venc_process(
____encoderHandle__________,     // handle to encoder
____cBuf___________________,     // buffer being input into encoder
_____encBuf________________,     // buffer output by encoder
```

Similarly, here's the prototype of the DMAI video decoder process function:

```
int Vdec_process(Vdec_Handle encoderHandle, Buffer_Handle inputBuffer,
                                            Buffer_handle outputBuffer);
```

```
Vdec_process(
____________________________,     // handle to decoder
____________________________,     // buffer being input into decoder
____________________________,     // size of buffer being put into decoder
```

> As a side note, the *while()* loop in `audio_thread.c` is conceptually similar to what we're doing here (get data → encode → decode), but the calls to the audio and video drivers are a little bit different – necessitating the changes to our *while()* loop in `video_thread.c`.

14. After *while()* exits, add cleanup code for the buffers and instances you added to this file.

There already is a "cleanup" section in *audio_thread_fxn* with two calls to close the input, and output video drivers. You need to add code to clean the remaining resources we have added:

– Video Encoder

– Video Decoder

– Engine

– Encoded Buffer

Hints:

- To view the prototypes for each of the three cleanup functions, you would normally refer to the documentation. We have provided them here for your easy reference:

```
Engine_close( Engine_Handle engineHandle );
Vdec_delete(  Vdec_Handle decoderHandle  );
Venc_delete(  Venc_Handle encoderHandle  );
```

- Don't forget that the engine cleanup must occur after all encoders and decoders associated with the engine have already been deleted.
- If in doubt, check out the cleanup code in audio thread file.
- Since we allocated the intermediate processing buffer (encBuf) using the DMAI *Buffer_create()* function, we recommend using its counterpart, *Buffer_delete()*, to free the buffer and release the memory.

15. Save, then exit, the `video_thread.c` file.

Modify video_thread.h

16. Extend the *video_thread_env* structure, adding an element to pass the engine name from *main()* to *video_thread_fxn()*.

This data structure is defined in the header file, and referenced by both `main.c` and `video_thread.c`.

You may remember our chapter discussion strenuously suggesting that the engine string name be passed to each thread, as opposed to opening the engine in main and passing a handle. To this end, our structure's new element should be a string named *engineName*.

```
char *engineName;
```

If needed, you can reference `audio_thread.h` to see how we extended the environment structure there.

Modify main.c

17. Fix the initialization of the *video_env* global variable.

After changing the definition of *video_thread_env* in a previous step, our initialization is now incomplete. We recommend initializing the new element to `NULL;`. The next step sets it to the engine's string name.

18. Modify `main.c` so that it passes the *engineName* to our video thread function.

Set the element you just modified, in *video_thread_env*, to the correct engine name.

What value should be used for the engine name?

Similar to the encoder and decoder string names we have used, you can find the engine string name defined in the engine configuration (.cfg) file. To make things easy, though, we have already #defined a constant for you:

```
#define  ENGINE_NAME = "encodedecode".
```

In our solutions we set the *engineName* field in the *video_env* structure right before we called the video thread's *launch_pthread()* function. It could have been done almost anywhere before this call, but this is location we chose.

Build, install and Run Application

19. Review the build script `makefile_profile.mak`, then build the program and test it out.

```
gedit makefile_profile.mak &
```

Notice that some additional –i paths have been added to the file; these options tell the compiler where to find the various header files for the code bundled into the "engine" library. *Keeping track of these headers is a bit tedious, but luckily, when we use Configuro again in the next chapter, it will automatically handle this for us.*

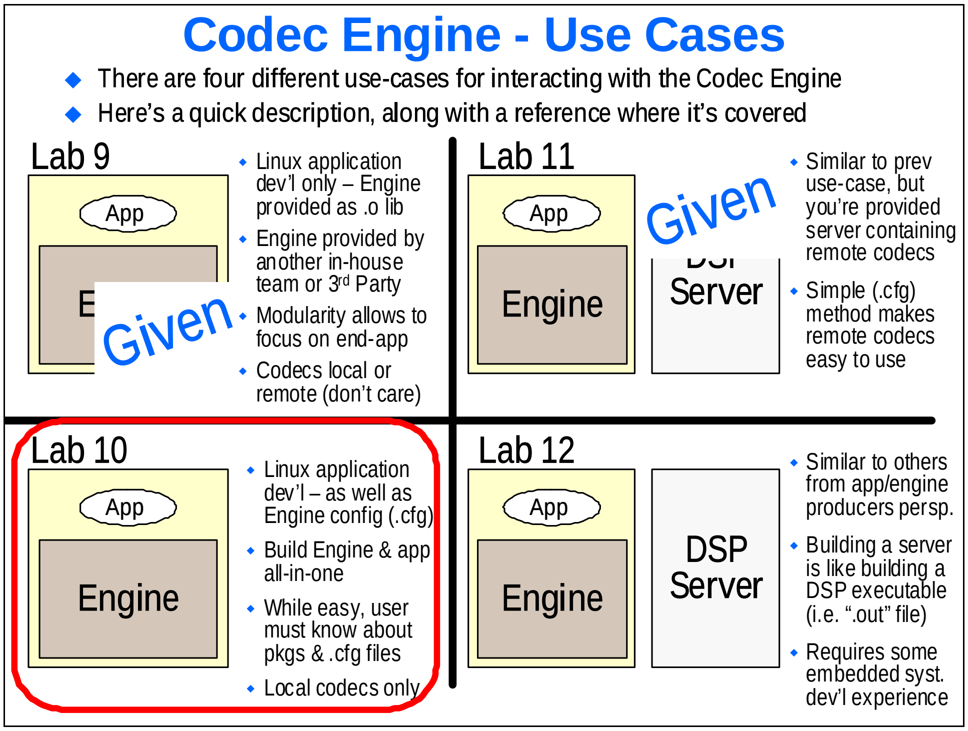
```
Close the file
Run gMake to build the application, then run it
```

Make sure your audio/video source is playing – you should successfully see and hear the results…

We now have the "framework" for using a real local codec in our application, even though the codec we're currently just a "copy codec". In future labs, we'll replace the dummy, copy codec with a real codec – and with very few modifications…

Lab 10 - Building an App With Engine

Lab 10 Context



Chapter Topics

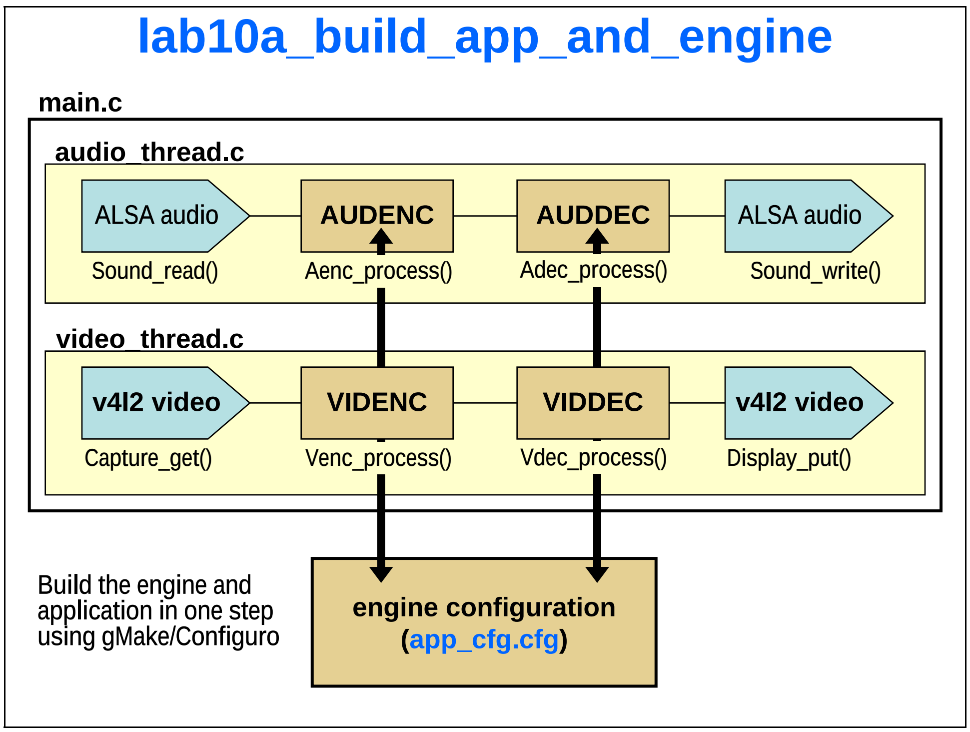
Lab Introduction

In this lab, you will extend Lab 9 by building the Engine (that was previously given to you) along with your application. After using Configuro in Lab 5, and your having used Engine in Lab 9, you should find this next step of building an engine pretty easy. In this exercise, our engine will still use the dummy (i.e. copy pass-thru) audio and video codecs that come with the Codec Engine examples, running locally on the ARM processor.

Once we copy our previous files into our new project folder, only two items must be added/changed to build our **own** engine along with our application.

a. Create a configuration (`app_cfg.cfg`) file that tells Configuro which codec packages you want in your engine.

b. Modify the XDCPATH (i.e. Configuro's search path) inside the makefile_profile.mak so Configuro can find the packages you want to use



Lab10a_build_app_and_engine

File Management

1. **Clean your `lab09b_use_published_engine_av` project.**

```
# cd /home/user/labs/lab09b_use_published_engine_av/app

# make clean
```

2. **Copy the `lab09b_use_published_engine_av` project to the Lab 10a directory.**

```
# cd ~/labs/lab10a_build_app_and_engine/app

# cp –R -f ~/labs/lab09b_use_published_engine_av/* .
```

. Note, if your previous Lab 9b didn't work, please copy from the solutions folder instead:

```
# cp –R -f ~/solutions/lab09b_use_published_engine_av/* .
```

3. **Copy our makefiles from Lab8b into lab 10.**

We need to go back to our original makefiles, since Lab 9 used a slightly modified makefile in order to provide gcc with the header/library paths needed for building with the *provided engine*. This lab once again relies on Configuro to provide the **l**ibrary (-l) and **i**nclude (-i) file path statements. (*You might remember us discussing this feature of Configuro in Chapter* 5.)

```
# cp -f ~/labs/lab08b_audio_video/app/makefile* .
```

Create RTSC configuration (.cfg) file

Our configuration file for building an engine containing only *local* codecs must specify three group(s) of packages. After creating the configuration file itself (in step 4), the following three steps outline how to import – and configure – each of the necessary packages.

4. **Open/create the RTSC configuration (`app_cfg.cfg`) file for editing.**

```
cd /home/user/lab/lab10a_build_app_and_engine/app

gedit app_cfg.cfg &
```

5. **You should already have the following RTSC modules/packages imported:**

```
var osalGlobal = xdc.useModule( 'ti.sdo.ce.osal.Global' );
var dmai = xdc.loadPackage( 'ti.sdo.dmai' );
var myDisplay = xdc.loadpackage ( 'ti.tto.myDisplay' );
```

Additionally, the osalGlobal module should be configured to osalGlobal.LINUX

None of this code should be changed, we will be appending to this file in the following.

6. **Import the four codec's we want to include in our *engine*.**

   In order to instantiate and use codecs in our C program, we need to specify them here so that Configuro will add them to our *engine*. Refer to the presentation to figure out the syntax, but here is a list of the codecs we plan to use:

   ```
   ti.sdo.ce.examples.codecs.viddec_copy.VIDDEC_COPY
   ti.sdo.ce.examples.codecs.videnc_copy.VIDENC_COPY
   ti.sdo.ce.examples.codecs.auddec_copy.AUDDEC_COPY
   ti.sdo.ce.examples.codecs.audenc_copy.AUDENC_COPY
   ```

   As a side note, it is common convention for package names to begin with your company and group name. In this case, ti.sdo.ce stands for:

   Texas Instruments . Software Development Organization . Codec Engine team

   The remaining part of the name was used to distinguish one package from another. In this case, you can see that we are including the codec examples provided by the CE team. As was the case in Labs 09 – 12, we use these dummy copy codecs; they simply perform a memcpy() inside the codec. While this makes them a bit un-exciting, they are great placeholders until we swap them out for real codecs. (In Lab 12b, we replace the video copy codecs with a real H.264 codec.)

   . One last item to note, again it is common practice for codec authors to use all CAPS for the actual module name inside a codec package. As a user, you just need to refer to the vendor's documentation (or examples) to figure out which name to include in your .cfg file.

7. **Create the actual engine, by importing the Codec Engine package, and configure it to include our codecs.**

   Once again, we refer you to the chapter discussion to figure out the module name and syntax for creating an engine. To provide consistency, though, we recommend that you use these names for your engine and codecs:

   Engine name: "encodedecode"

   videnc copy: "video_encoder" (local)

   viddec copy: "video_decoder" (local)

   audenc copy: "audio_encoder" (local)

   auddec copy: "audio_decoder" (local)

8. **Check your work**

   You can compare app/app_cfg.cfg to the provided engine/engine_cfg.cfg that was provided with the engine files of lab09b. Your app_cfg.cfg should have everything that was previously included (as per step 5) appended with the items you see in engine_cfg.cfg (as per steps 6 and 7).

9. **When complete, save and close your config file.**

10. Remove engine directory

Now that the engine packages have been added to app_cfg.cfg, we no longer need the provided engine_DEBUG.ov5T and engine_RELEASE.ov5T files.

```
# rm –Rf /home/user/labs/lab10a_build_app_and_engine/engine
```

Update XDCPATH

11. Find the XDCPATH definition in your `makefile_profile.mak` file.

```
gedit makefile_profile.mak &
```

12. Examine Configuro search path for the packages specified in your .cfg file.

You might remember we created a gMake variable (XDCPATH) which tells Configuro where to search. For simplicity, the makefile provided in lab07 has already been configured to search the codec engine repositories (even though these paths have not been required previous to this lab.)

In `makefile_profile.mak`, you can confirm that Configuro will search the following repositories (i.e. directories) which contain the packages we included with our .cfg file:

ti.sdo.ce.engine	Codec Engine	`$(CE_INSTALL_DIR)/packages`
‘copy’ codecs	CE Examples	`$(CE_INSTALL_DIR)/examples`
req by codecs	xDAIS	`$(XDAIS_INSTALL_DIR)/packages`
req by CE	CMEM	`$(CMEM_INSTALL_DIR)/packages`
req by CE	Contig Mem Alloc	`$(FC_INSTALL_DIR)/packages`

Build, Install, Run (and hopefully not need to debug…)

13. Make and install your program to the DVEVM target.

14. On the DVEVM board, run the `loadmodules.sh` script if it has not been run since the board was last booted.

> **Hint:** If you are unsure whether or not the `loadmodules.sh` script has been run, you can always run the `unloadmodules.sh` script and then re-run the loadmodules.sh script to put the system into a known state.

15. Execute the `app_debug.xv5T` application.

Make it fail

Once you have your program working, it's a good idea to figure out what it looks like when you make a mistake. A majority of all build mistakes are caused by incorrect path statements. For example, if you don't specify the correct search paths, Configuro will fail. Actually, this is a good thing; it is much better to fail early during build, than later during runtime.

We recommend that if you didn't accidentally get a failure when first building and running your program that you force an error and look at its affect.

16. Open your `makefile_profile.mak` and modify the XDCPATH statement – remove the CMEM directory reference – then save the file.

```
gedit makefile_profile.mak &
```

17. Upon rebuilding, without the CMEM reference, you should see this error:

```
4.  ----- Starting Configuro for app_cfg.cfg  (note, this may take a minute)

js: "/home/user/dvsdk_1_30_00_40/xdctools_3_10/packages/xdc/cfg/Main.xs", line 193:
xdc.services.global.XDCException: xdc.PACKAGE_NOT_FOUND: can't locate the package
'ti.sdo.linuxutils.cmem' along the path:
'/home/user/dvsdk_1_30_00_40/codec_engine_2_00_01/packages;/home/user/dvsdk_1_30_00_4
0/codec_engine_2_00_01/examples;/home/user/dvsdk_1_30_00_40/xdais_6_00_01/packages;/h
ome/user/dvsdk_1_30_00_40/dsplink_140-
05p1/packages;/home/user/dvsdk_1_30_00_40/framework_components_2_00_01/packages;/home
/user/dvsdk_1_30_00_40/xdctools_3_10;/home/user/dvsdk_1_30_00_40/xdctools_3_10/packag
es;/home/user/dvsdk_1_30_00_40/xdctools_3_10/packages;/home/user/workshop/lab10_build
_engine/app/DEBUG/app_cfg/./..;'. Ensure that the package path is set correctly.
    "/home/user/dvsdk_1_30_00_40/xdctools_3_10/packages/xdc/cfg/Main.xs", line 154
    "/home/user/dvsdk_1_30_00_40/xdctools_3_10/packages/xdc/xs.js", line 160
gmake: *** [package/cfg/app_cfg_x470MV.c] Error 1
```

While this error does look intimidating, it does contain the necessary information we need to decipher, and solve, this problem. Look for this key item which leads us to our solution:

`can't locate the package '`**`ti.sdo.linuxutils.cmem`**`'`

In this case, the package name gives us a good place to start looking for a solution. When we see ***cmem***, it makes it pretty easy to track down the problem. If you look thru the path Configuro is searching, you should notice that the CMEM directory is missing. (Of course, because we just deleted it to force this error.)

So, once we know this error, we need to find the correct directory to reference on the XDCPATH string. With a little searching, you should be able to find the path. Look thru the CMEM directory, until you find the folder that contains the path that error referenced:

`ti/sdo/linuxutils/cmem`

The folder that holds "ti" from the above path needs to be added to the XDCPATH. In our VMware image (at the time of this printing), the path should be:

```
/home/user/dvsdk_2_00_00_22/codec_engine_2_23_01/cetools/packages
```

Now, if you remember that we have put all of our hardcoded path references in an imported file called `setpaths.mak`, then you can get away with simply using:

```
$(CMEM_INSTALL_DIR)/packages
```

. You can try out both of these to assure yourself they both work.

Note: Notice how package names correlate to a filesystem. Whenever you see a "." in a package name, know that it will represent a directory level in the containing filesystem. With a little practice, figuring out these problems should become less daunting.

---

18. Repair your XDCPATH and re-test your solution.

(Optional) Lab10b _engine_deliverable

File Management

19. Examine the provided files in `lab10b_engine_deliverable/engine`.

```
# cd /home/user/labs/lab10b_engine_deliverable/engine
# ls
```

```
You should see the following files:
```

```
audio_decoder_dummy.c     audio_encoder_dummy.c
video_decoder_dummy.c     video_encoder_dummy.c
engine_dummy.c            engine_cfg.cfg
makefile                  makefile_profile.mak
```

20. Examine `audio_decoder_dummy.c`.

The “dummy” files are all conceptually the same. You will see one function which exercises each of the four codec engine calls for a given module, i.e. create, control, process and delete. This forces these functions to be included in the engine deliverable from the codec engine libraries that the Configuro tool will link.

21. Examine `makefile_profile.mak`.

Locate the “build engine deliverable” comment preceding the rule:

```
$(PROGNAME)_$(PROFILE).ov5T : (…)
```

Within this build rule is the link rule to link the $(C_OBJS), which in this case are the dummy files, with the libraries provided by Configuro, $(PROFILE)/$(CONFIG)/linker.cmd

Note that the link command contains the flags:

`-Wl,-r`	Forces a partial build, i.e. .o instead of an executable
`-nostdlib`	Don’t link the standard libraries (will be linked by app)

22. Copy the files from `lab09b_use_published_engine`.

This copy will cause the engine directory to be overwritten, but the new ‘dummy’ files should remain untouched.

```
# cd /home/user/labs/lab10b_engine_deliverable
# cp –R ../lab09b_use_published_engine/* .
```

23. Rebuild the engine deliverable.

```
# cd engine
# make clean
# make all
```

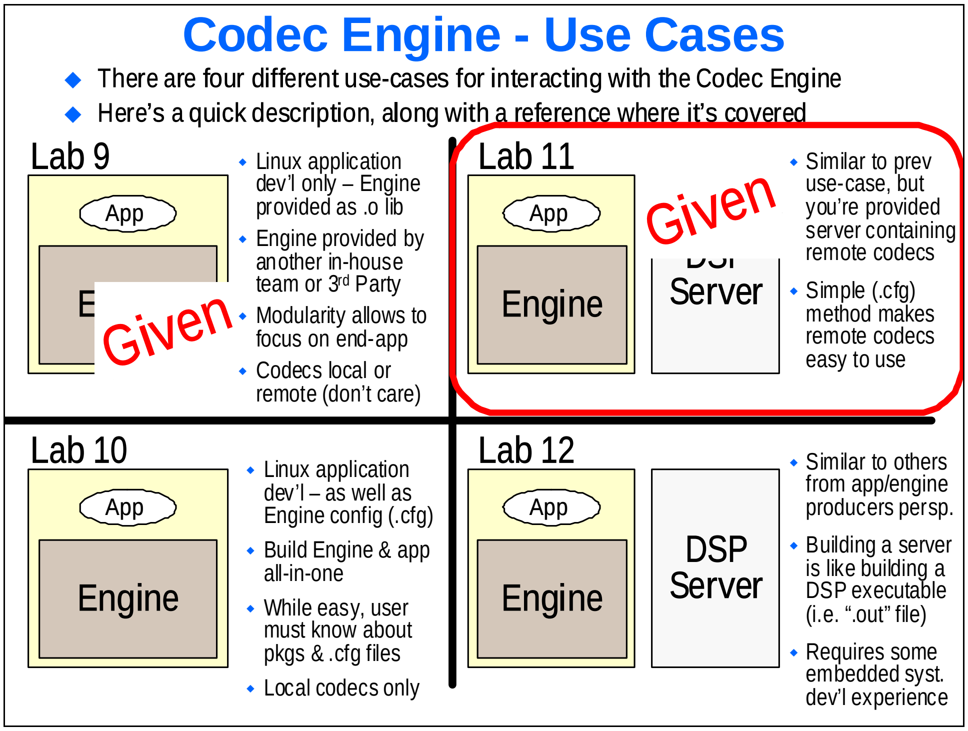
24. Rebuild and install the application.

```
# cd ../app
# make clean
# make install
```

25. Test the application.

Remote Codecs: Given a DSP Server

Lab Context



Lab Outline

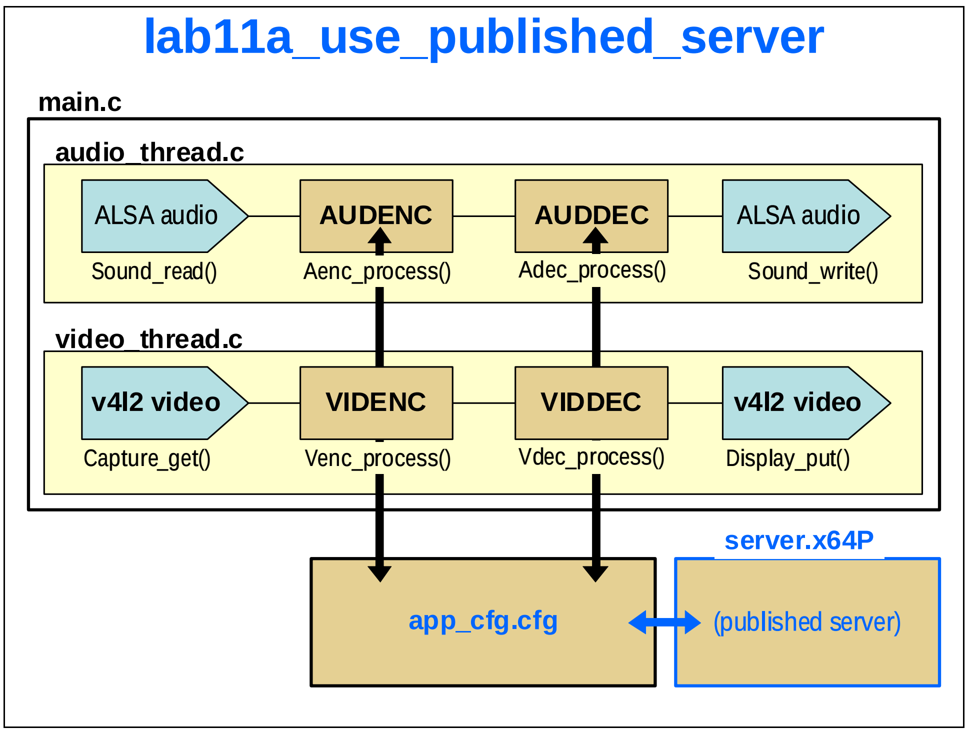
Lab 11 Introduction

This exercise introduces DSP-based remote codecs. You will be provided a "published" DSP server (i.e. DSP executable program). By modifying your Configuration (.cfg) file, your application's encode/decode functions will now run over on the DSP.

No application code needs to be changed to call remote vs. local (ARM-based) codecs. Though, we will change how we allocate our memory buffers – to be sure they are allocated contiguously within Linux.

Finally, you will want to tweak your makefile's "install" rule to copy over the server file along with your executable application.

In Lab 11b you will experiment with real H.264 codecs; and in the next lab, you will use the Codec Engine wizards to build your own DSP server.



Lab 11a – Using a Published Server

Prepare/copy project files

1. **Copy the contents from our last lab into the `lab11a_publish_server`.**

```
cd ~/labs/lab11a_user_published_server
cp –R -f ~/labs/lab10a_build_app_and_engine/* .
```

2. **Locate the "published" server in the lab11a server folder.**

   This lab exercise uses the server that you'll be creating in the next lab. For your convenience, we've copied the server files over to the server directory.

```
cd ~/labs/lab11a_user_published_server/server
```

3. **Examine `package.xdc` for the *server* package.**

   The package declaration:

```
package server [1, 0, 0] {

}
```

   Indicates that this is the `server` package, revision 1.0.0, and that it has no modules – the module declarations would appear inside the open and close braces. *(I guess we weren't very creative with our name of **server**, but we wanted to keep it simple.)*

4. **Examine the `codec.cfg` XDC configuration file for our *server* package.**

   You should see four *xdc.usemodule()* statements that import the four codecs that we have been using thus far (though, since this is the server cfg file, these end up being DSP versions of the codecs): AUDDEC_COPY, AUDENC_COPY, VIDDEC_COPY and VIDENC_COPY

   Further down in the server.algs[] array, you can see the codecs were assigned the names "viddec_copy," "videnc_copy," "auddec_copy" and "audenc_copy."

5. **Locate the server executables.**

```
# ls -l bin/
```

   You should see `server.x64P`. Therefore the path of the server executable – inside the server's package is:

.   ____

Modify Engine Configuration File (app_cfg.cfg)

6. **Change back to your application directory**

```
# cd /home/user/labs/lab11a_build_app_and_engine/app
```

7. **Open the config file (`app_cfg.cfg`) and update the OSAL runtime environment to include DSPLINK.**

```
osal.runtimeEnv = ________________
```

8. **Modify the Engine.create() method to use the new "Create From Server" feature of the Codec Engine.**

We could have listed each server codec individually, but to make it easier, as well as less error prone, we recommend using the new Codec Engine method which extracts all the required information from the server's package.

Referring to the chapter's .cfg example, replace the **Engine.create()** method with the new **Engine.createFromServer()** method.

Replace

~~var demoEngine = Engine.create("encodedecode", [~~
~~{name: "video_encoder", mod: VIDENC, local: true},~~
~~{name: "video_decoder", mod: VIDDEC, local: true},~~
~~{name: "audio_encoder", mod: AUDENC, local: true},~~
~~{name: "audio_decoder", mod: AUDDEC, local: true},~~
~~]);~~

With:

```
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEngine = Engine.createFromServer(
    "<engine_name_here>",          // Engine name (as referred to in the C app)
    "<server_exec>",               // Where to find the .x64P exe, inside the server's package
    "<server_package_name>"        // Server's package name
);
myEngine.server = "<server_exec>"; // Loc'n of server exe at runtime, relative to .xv5T program;
                                   //   only needed if not found in the same folder as . xv5T
```

Hints:

- The engine name can be whatever you want, but it should match the name used in your applications *Engine_open()* call. So far, we've been using "***encodedecode***".
- You determined the server executable name and path relative to server package directory in step 0.
- You determined the package name of the server in step 3.
- Because the server `.x64P` and the ARM `.xv5T` applications will both be executed from the same `/opt/workshop` directory, the engine's *.server* property does not need to be specified. (You can leave this line out). Or, you can specify it with the same executable name as used above, but with no relative path (i.e. `"./server.x64P"`)

Changes needed to makefile_profile.mak

Two changes are required to get our build script up-to-date.

9. We must add an additional directory path or two to our XDCPATH variable.

Here are a couple hints to help you complete this step:

- Now that we're using remote codecs – and setting osal=DSPLINK_LINUX – we need to make sure Configuro can find the path to the DSPLINK package.
- We need to tell Configuro where our server package is located, so we need to specify its repository path. A good rule of thumb is that the package build directory is always located at:

  <Repository_Path>/<package_name>

  Where in the package name, the periods '.' are replaced by forward slashes '/'
  Hence, if the server package name is `ti.sdo.ce.examples.servers.all_codecs`, the path would be: `/home/user/ti-dvsdk_omap3530-evm_4_00_00_17/codec-engine_2_25_05_16/examples`.

  Then again, that wasn't the name (or path) for our package. What is our server packages path? ____

- We also need to specify the paths to our codecs. The 'copy' codecs we've been using thus far are located in the Codec Engine examples directory.
- Since this usually becomes an exercise in typing rather than learning, these paths have already been added to `makefile_profile.mak`. Locate the XDCPATH variable and verify that it contains our package locations. Which three paths on the XDCPATH assignment represent the two packages we've talked about here:

  1. ____

  2. ____

  3. `$(CE_INSTALL_DIR)/examples` (we decided to give you one of them)__________

10. Modify the "install" rule so that it also copies the server executables.

We handle the server install similar to how we added the OSD file to our make install rule in Chapter 8. Again, this works since install rule copies all dependencies to the $(EXEC_DIR).

```
PROGNAME          := app
CONFIG            := app_cfg
INSTALL_OSD_IMAGE := ../osdfiles/ti_rgb24_640x80.bmp
INSTALL_SERVER    := ________________________

.PHONY  : install
install : $(PROGNAME)_$(PROFILE).xv5T $(INSTALL_OSD_IMAGE) $(INSTALL_SERVER)
            @echo ...
```

Modify audio and video "_thread.c" files

11. Verify that buffers which are shared between Arm and DSP are contiguously allocated.

Remember, the ARM device's memory management unit (MMU) allows it to remap non-contiguous memory buffers into contiguous memory buffers. This is accomplished using virtual addresses. Since the C64+ DSP does not have an MMU (*or even when the DSP has one, due to the need for speed, it doesn't want to use memory virtualization*), any buffers passed by the ARM to the DSP need to be allocated as physically contiguous buffers. For this reason, we want the shared buffers to be allocated with the Linux CMEM (contiguous memory) driver.

The DMAI *Buffer_create()* call can either allocate physically contiguous or virtually in contiguous (which may or may not also be physically contiguous) memory depending on the attributes structure it is provided. The default memory attributes structure specifies contiguous memory, so all of the dynamic memory allocations in our applications are already contiguous. Bottom line, no change needs to be made.

For more information on the memory functions (as well as VISA functions), you can go to the Codec Engine documentation by using:

```
cd /home/user/ti-dvsdk_omap3530-evm_4_00_00_17/codec-engine_2_25_05_16
mozilla file://$(pwd)/codec_engine_2_25_05_16_ReleaseNotes.html
```

Select the *Documentation* link from the top of the release notes, then select *Codec Engine Application Programming Interface (API) Reference Guide* in html, then select the Memory link from the bottom of the page that comes up.

Build and run

12. Build the application.

```
make debug
```

Did it build correctly? Why not?

. ____

. ____

Hint: look back to step 4.

13. Install and Run your application.

If you're rebooted the board recently, don't forget to load the DSPLINK and CMEM modules.

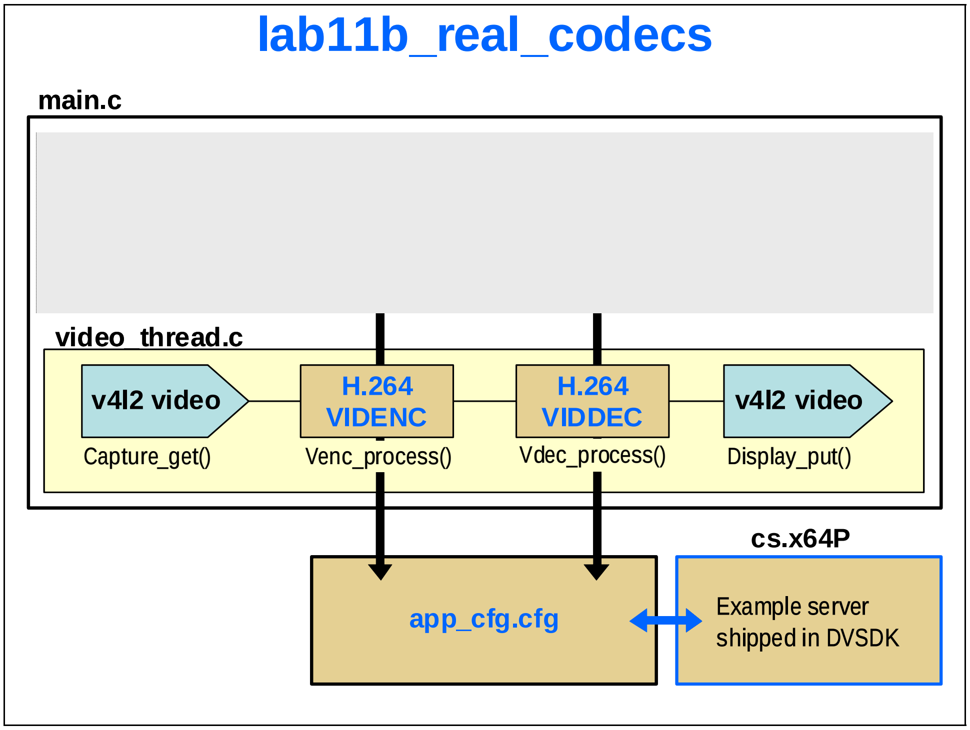
```
loadmodules.sh
```

Then, execute the `app_DEBUG.xv5T` application.

Lab 11b – Using Real H.264 Codecs

In Lab 11b you will swap out the dummy *video_copy* codecs shipped in the Codec Engine, for real H.264 codecs that are part of the SDK.

Since we want to concentrate on the video codecs, we'll disable the audio thread in our `main.c` file, leaving us with running just the video and osd threads.



Prepare/copy project files

14. Copy the *app* and *osdfiles* directories from `lab11a_publish_server`.

```
cd ~/labs/lab11b_real_codecs

cp –R -f ~/labs/lab11a_publish_server/app .

cp –R -f ~/labs/lab11a_publish_server/osdfiles .
```

Did we forget to copy something? No – since this lab uses a DSP server (`cs.x64P`) that ships with the DVSDK, we don't need a server folder in our `lab11b` directory. We'll just update our `makefile_profile.mak` to point to the new repository.

Modify Engine Configuration File (app_cfg.cfg)

15. Change to the application directory.

```
# cd app
```

16. Open the config file (`app_cfg.cfg`) and update the "Create From Server" method.

We could have listed each server codec individually, but to make it easier, as well as less error prone, we recommend using the new Codec Engine method which extracts all the required information from the server's package.

Once again, here's the "prototype" for *.CreateFromServer()*:

```
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEngine = Engine.createFromServer(
    "<engine_name_here>",        // Engine name (as referred to in the C app)
    "<server_exec>",             // Where to find the .x64P exe, inside the server's package
    "<server_package_name>"      // Server's package name
);
myEngine.server = "<server_exec>"; // Loc'n of server exe at runtime, relative to .xv5T program;
                                   //   only needed if not found in the same folder as . xv5T
```

We need to update our previous .cfg file with the information for the DVSDK's server:

- **Build-time Server Package:** `ti.sdo.server.cs`
- **Build-time Path to executable:** `./bin/cs.x64P`
  (relative to package)
- **Run-time Server location:** `./cs.x64P`
  (relative to ARM executable)

Changes needed to makefile_profile.mak

Two changes are required to get our build script up-to-date.

17. We must add an additional directory path to our XDCPATH variable.

We need to add the location of the new codecs and server to our XDCPATH so that Configuro can find them. They repository where they're located is:

```
$(SDK_INSTALL_DIR)/codecs-omap3530_1_01_00/packages
```

We recommend you verify the server package listed in the previous step is found along this repo path.

18. Modify the "install" rule so that it also copies the server executable.

We handle the server install similar to how we added the OSD file to our make install rule in Chapter 8. Again, this works since install rule copies all dependencies to the $(EXEC_DIR).

```
PROGNAME          := app
CONFIG            := app_cfg
INSTALL_OSD_IMAGE := ../osdfiles/ti_rgb24_640x80.bmp

INSTALL_SERVER    := __________________________________________________

...

.PHONY  : install
install : $(PROGNAME)_$(PROFILE).xv5T $(INSTALL_OSD_IMAGE) $(INSTALL_SERVER)
             @echo ...
```

What should the server location be? Well, what path did you find the .x64P file when you verified it existed in the previous step?

Modify main.c file

19. Comment out the audio pthread create call.

As we stated earlier, we just want to concentrate on the video parts of the lab. To that end, we recommend you comment out the audio thread create call.

```
/* Create a thread for audio */
/* Commenting out audio thread -- this lab focusing video codecs
    DBG( "Creating audio thread\n" );

    audio_env.engineName = ENGINE_NAME;

    if( launch_pthread( &audioThread, REALTIME,
                        sched_get_priority_max( SCHED_RR ),
                        audio_thread_fxn,
                        (void *) &audio_env )
            != thread_SUCCESS )
    {
        ERR( "Failed to create audio thread\n" );
        status = EXIT_FAILURE;
        video_env.quit = 1;
        osd_env.quit = 1;
        goto cleanup;
    }

    initMask |= AUDIOTHREADCREATED;
*/
```

Edit/Replace video_thread.c file

20. Edit – or replace – the `video_thread.c` file.

The changes required to `video_thread.c` are not difficult, but some of them are a bit tedious. We’ll review them here, and then give you the option to make the edits … or copy the solutions `video_thread.c` into your labs folder.

```
/* DMAI headers */
#include <ti/sdo/dmai/Dmai.h>
...
#include <ti/sdo/dmai/BufferGfx.h>
//#include <ti/sdo/dmai/ce/Vdec.h>      // As discussed briefly in
//#include <ti/sdo/dmai/ce/Venc.h>      // chapter 9, many new video
#include <ti/sdo/dmai/ce/Vdec2.h>       // codecs use a later
#include <ti/sdo/dmai/ce/Venc1.h>       // version of the XDM standard
=====================================
/* Video encoder and decoder used */
//#define VIDEO_ENCODER "videnc_copy"   // When the DVSDK team built
//#define VIDEO_DECODER "viddec_copy"   // their server, they used
#define VIDEO_ENCODER "h264enc"         // these names for their
#define VIDEO_DECODER "h264dec"         // codecs
=====================================
Venc1_Handle          encoderHandle = NULL;    // handle to video encoder
Vdec2_Handle          decoderHandle = NULL;    // handle to video decoder
VIDENC1_Params        eParams       = Venc1_Params_DEFAULT;
VIDDEC2_Params        dParams       = Vdec2_Params_DEFAULT;
VIDENC1_DynamicParams eDynParams    = Venc1_DynamicParams_DEFAULT;
VIDDEC2_DynamicParams dDynParams    = Vdec2_DynamicParams_DEFAULT;

Int                   ret           = Dmai_EOK;
=====================================
/* Setup Display */
```

Changing to VIDENC1 & VIDDEC2 means these change, too.

The Display driver initialization needs to move after the Codec Engine (and codecs) setup code. Since we're sharing buffers with the decoder – and the we interrogate the decoder for "bufSize", the setup display section must be relocated (in our solutions, it's right before the while() loop.

```
=====================================
    eParams.inputChromaFormat = XDM_YUV_422ILE;
    eParams.maxBitRate        = 4000000;
    eParams.maxWidth          = gfxAttrs.dim.width;
    eParams.maxHeight         = gfxAttrs.dim.height;

    eDynParams.targetBitRate  = 4000000;
    eDynParams.inputWidth     = gfxAttrs.dim.width;
    eDynParams.inputHeight    = gfxAttrs.dim.height;

    encoderHandle = Venc1_create(…)
  =====================================
    dParams.forceChromaFormat    = XDM_YUV_422ILE;
    dParams.maxWidth             = VideoStd_D1_WIDTH;
    dParams.maxHeight            = VideoStd_D1_PAL_HEIGHT;

    decoderHandle = Vdec2_create(…)
=====================================
// DMAI will assume that the output buffers provided to the decoder are
//     at least the size requested by the decoder, so for consistency
//     it is good practice to always use a decoder getOutBufSize query
//     to determine the size of the buffer table buffers
bufSize = Vdec2_getOutBufSize( decoderHandle );
hBufTabDecoderDisplay = BufTab_create(..., bufSize, ...);

Vdec2_setBufTab( decoderHandle, hBufTabDecoderDisplay );

    initMask |= VIDEODECODERCREATED;

/* Set buffer size for intermediate buffer (for encoded data)          */
/* as the size of a full frame                                         */
    //encBufSize = bufSize;
    encBufSize = Vdec2_getInBufSize( decoderHandle );
```

New params required for H.2664 encoder and decoder.

The definition of hBufTabDecoderDisplay now depends upon size specified by the video decoder.

Build and run

21. Build the application.

```
make debug
```

22. Install and Run your application.

Always remember, if you've rebooted the board recently, don't forget to load the DSPLINK and CMEM modules.

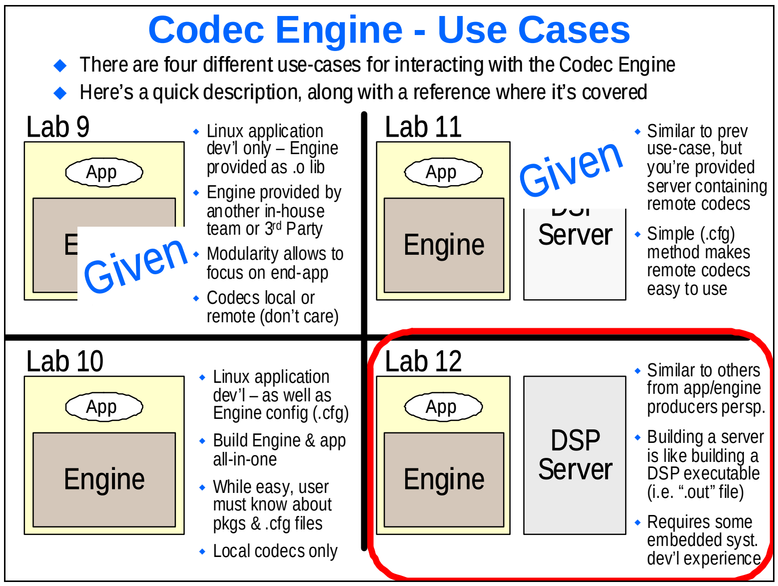
```
loadmodules.sh
```

Then, execute the `app_DEBUG.xv5T` application.

Lab 12 - Building a DSP Server

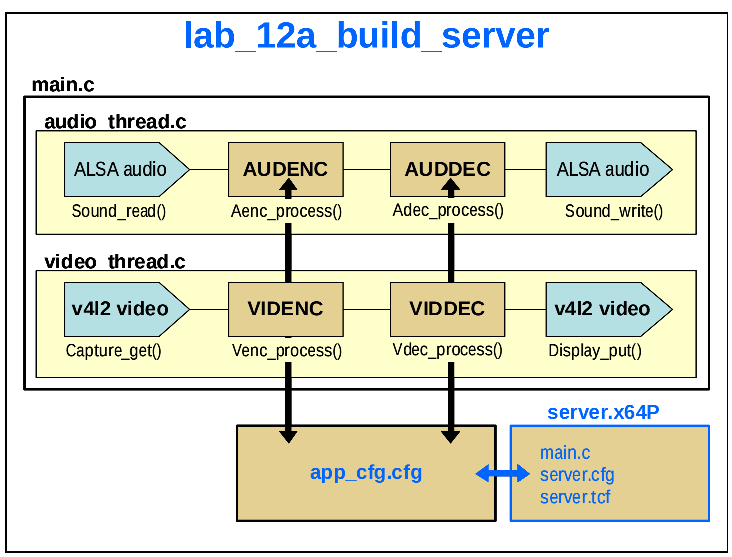
Where we're at in the Codec Engine lab flow

We have finally arrived at the final step in our exploration of the Codec Engine; that is, we are ready to build our own DSP Server. Lab12a_build_server focuses on this task. The optional exercise (Lab12c_h264), if you have time, directs you to change out the video copy-based codecs used in the last three labs for a real (watermarked) H.264 encoder and decoder.



Introduction

In this lab, you will extend Lab 11 to by letting you build the DSP server that was provided pre-built for you. You will do this by running the Codec Engine's DSP Server Wizard to create the files needed to configure and build a DSP server.



Lab Table of Contents

Lab12a_build_server

While we do not have time in this workshop to build a DSP server piecewise from the ground up (i.e. each file from scratch), this is unnecessary nowadays. Rather, the Codec Engine now provides a DSP Server Wizard to help us quickly create the necessary files.

We'll use this tool, modify one or two of its output files to suit our needs, then build the DSP Server to use along with our previous ARM/Linux application.

File Management

1. **Copy the `Lab11a_publish_server/app` directory over to `lab12a_build_server`.**

```
cd ~/labs/lab12a_build_server/app
cp –R -u ~/labs/lab11a_publish_server/app/* .
```

2. **Change to the `/home/user/labs/lab12a_build_server/osdfiles` and copy the files from your previous lab exercise.**

```
cd ~/labs/lab12a_build_server/osdfiles
cp –R -f ~/labs/lab11a_publish_server/osdfiles/* .
```

3. **Change into the Lab12a <u>server</u> directory and examine the new `makefile`.**

   This is where most of our work will take place in this exercise. There should only be one file in this folder to start with – `makefile` – which has been modified a bit in order to build a DSP server.

```
cd ~/labs/lab12a_build_server/server
gedit makefile_server.mak &
```

(set gmake source highlighting via gedit menu: *View→Highlight Mode→Sources→Makefile*)

. What command invokes the DSP Server Wizard? ____________________________________

. ____

. Where directory path do you find the DSP Server Wizard? ______________________________

. ____

. What tool do we use to build our DSP server executable (and package)? _________________

. ____

. Since our DSP server is a DSP executable (not just a library), why do we build it into a RTSC package? ____

. ____

To reiterate some of the differences in the makefiles:

- Similar to Configuro, we have added two more variables to run the Server Wizard and XDC tools:

```
GENSERVER := $(XDC_INSTALL_DIR)/xs ti.sdo.ce.wizards.genserver
MAKEPKG   := $(XDC_INSTALL_DIR)/xdc
```

- Check out the XDCPATH definition. One important path to note is where the copy-based codecs we are using are located: `$(CE_INSTALL_DIR)/examples`.

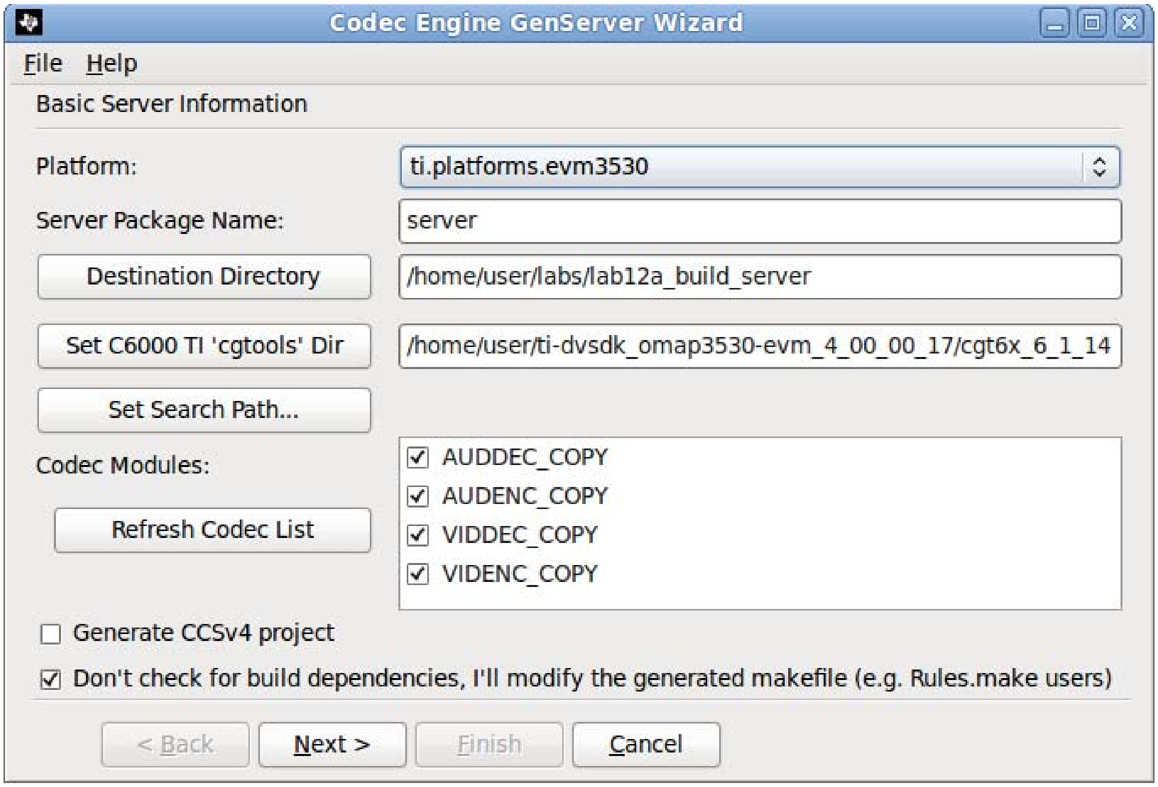
```
export XDCPATH:=$(CONFIG_BLD_PATH);$(CE_INSTALL_DIR)/packages; … ;$(XDCROOT)
```

- Finally examine the make rules we created to: **run the server wizard**, **build the server**, and **clean the server** directory. The *build* and *clean* rules make use of the XDC build tool; this is the easiest way to build the server application and wrap it in a RTSC package.

Running the DSP Server Wizard

4. Let's execute the rule we just examined to start the DSP Server Wizard.

```
make –f makefile_server.mak run_server_wizard &
```



5. Fill in the first dialog of the GUI DSP Server Wizard.

Note, it may take a minute or two for the Wizard to appear, this is normal since it is searching for any codecs/algorithms contained along the XDCPATH.

When it appears, fill in the necessary information:

Platform: `ti.platforms.evm3530` (to match our board)

Package Name: `server` (to match the name from our last lab)

Destination Dir: `/home/user/labs/lab12a_build_server`

C6000 Tools Dir: see above

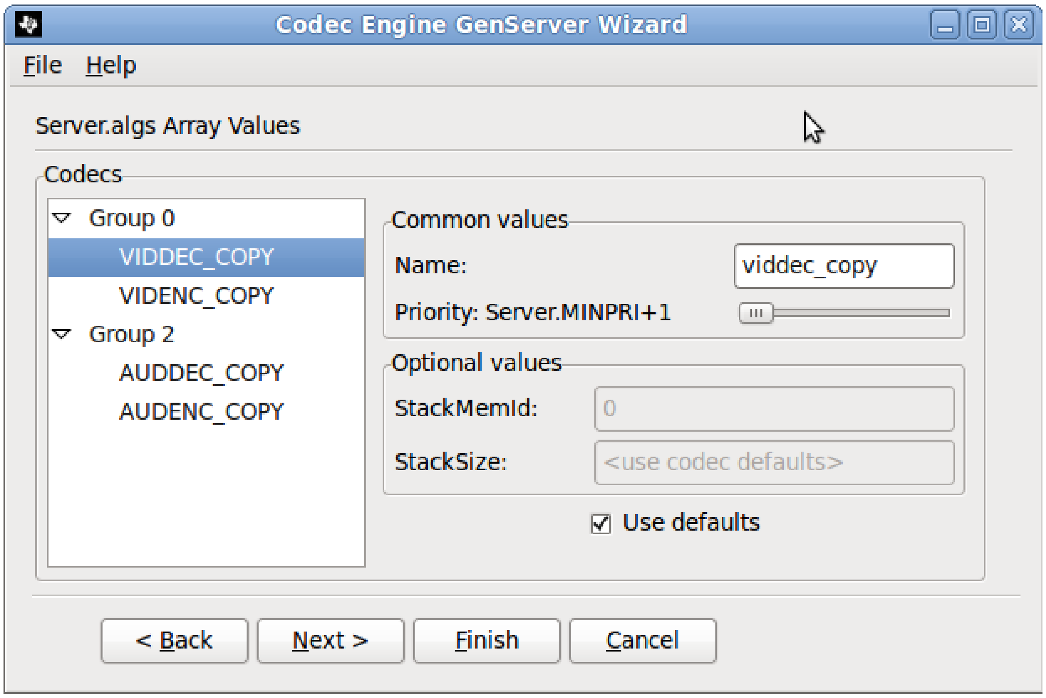
Codecs: Check the algo's we have been using in the last exercises:

```
AUDDEC_COPY
AUDENC_COPY
VIDDEC_COPY
VIDENC_COPY
```

There is no scrollbar for codec selection in this version of the wizard. Though, you can scroll down/up with the cursor keys.

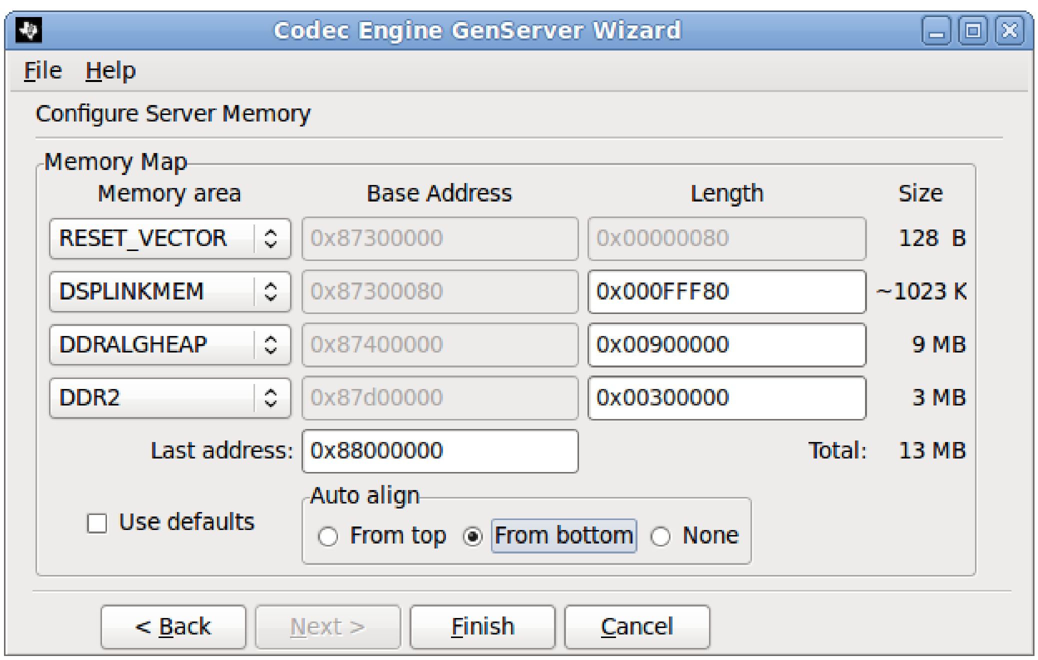
Set the **checkboxes** as shown at the bottom of the dialog.

6. Click *Next* for the second screen of the DSP Server Wizard.



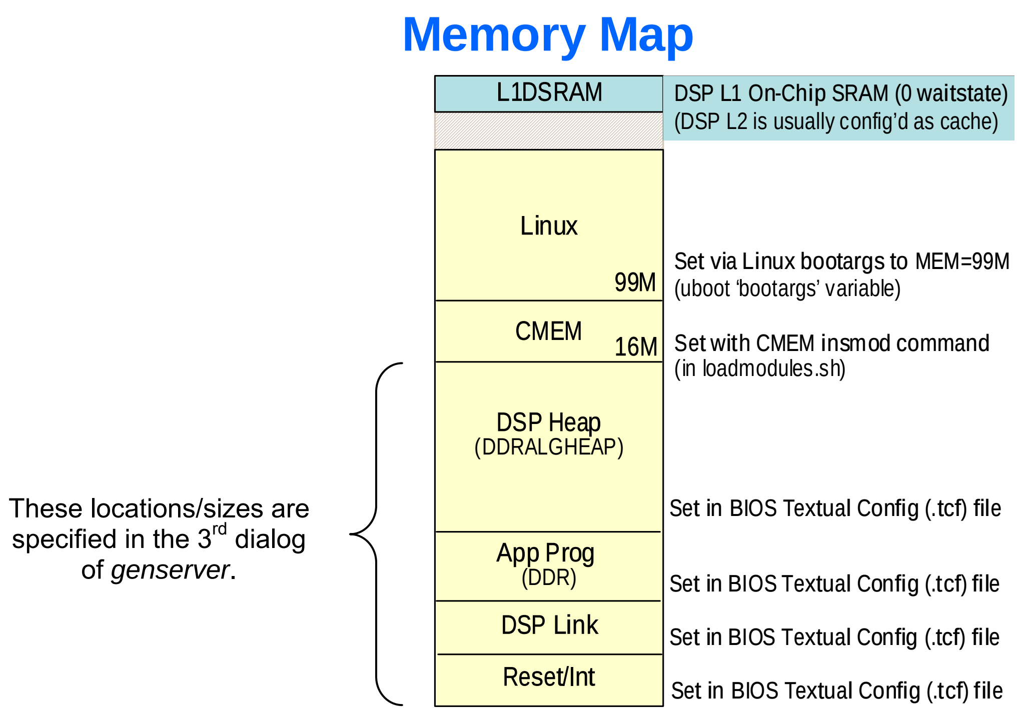
. We will use the default values for everything in the second step of the wizard.

7. Click *Next* for the third step



`Change your dialog to match the dialog settings we show above.`

The third dialog lets you set the location for memory usage for the DSP. These settings will be saved by the wizard into a BIOS textual configuration include file (`memmap.tci`).



Note

When building the labs, we originally chose the ***Use defaults*** setting. Doing this, we got an error that said the entire memory specified had to be divisible by 4K. In response, we changed the settings as provided here – modifying the DSPLINKMEM space slightly to compensate for the 80H allocated to the Reset/Interrupt vectors. *(The error message was well stated, but we hope the defaults will work out-of-the-box in the next version of the wizard.)*

8. Close DSP Server Wizard and save your entries.

Click Finish to close the server wizard.

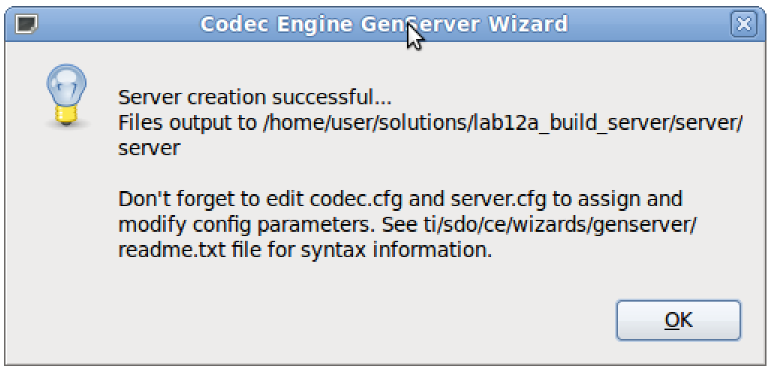
```
Finish
```

When it asks if you want to save the values entered into the server wizards dialog, go ahead and say yes. Save the file in a convenient location, for example, in the lab directory:

```
~/labs/lab12a_build_server/server
```

If you should need to re-run the Server wizard, you can easily re-load your answers by using the XML settings file.

Also, you will receive a note stating:



This dialog is letting you know that you can tune the server configuration by editing the `codec.cfg` file. In our case, though, we don't need to change any of these settings.

Note: You may receive one or two warnings when the wizard starts to write the server files. One, it will warn you that the target directory already has files in it. That's OK, in our case we've got the makefile located in the folder already. Also, we just saved the .xml file there.

The other warning is that some files may need to be edited. That doesn't apply for this lab exercise, but will for a future exercise.

Examine the Server Files
(created by the CE DSP Server Wizard)

9. Examine the files created by the wizard.

If you're not already in the `server` directory, switch over to it and examine the following files.

> ### Hint
> When opening and viewing the following .tcf and .cfg files with gedit, you may want to view the file with "javascript" syntax highlighting.
>
> ```
> View -> Highlight Mode -> Scripts -> JavaScript
> ```

server.tcf

This is a platform specific file, thus its contents vary slightly based on which platform you selected in the wizard. You are not expected to understand the details of this file, though it should be clear that it is used to configure the memory map of the DSP as well as creating and initializing various DSP/BIOS objects. Understanding the details of how this file configures the DSP/BIOS operating system is the subject of TI's 4-day BIOS workshop.

Notice, too, that this file (on line 9) imports the memory map settings in the file: memmap.tci.

server.cfg

This is another platform specific file. Note that the server configuration is similar to the engine configuration as performed in `app/app_cfg.cfg`. Additionally the configuration file configures the DMAN3 module, which is the module that provides DMA resources to server codecs, and the DSKT2 module, which is the module for providing memory to server codecs. (Note, DMAN3 will be discussed in a later chapter.) This file "imports" the codec.cfg file to obtain the array of codecs/algorithms you selected to be included the 'server'.

codec.cfg

This file should bring in the codecs and algorithms you specified during the server wizard. It also configures each algorithm module per the defaults specified in that module (if there were any), and then builds the array of algorithms using the name, priority, and other details you specified while running the wizard.

package.xdc

This file simply states the name of the package we are creating. It should reflect the name we provided in the first step of the server wizard GUI.

makefile

The server creates a makefile to build the server, but since it requires you to go in and edit the paths to our tools, we've decided to create our own. So ours isn't overwritten, we've named it makefile_server.mak

Other files …

For our lab exercise, you should not need to modify the remaining lab files. While there are times when one of these files may need to be edited – say, to access an advanced feature of RTSC packages – but this is not need for this lab.

Build the Server

10. Build the server package.

Using the makefile again, run XDC to build the server.

```
make –f makefile_server.mak build_server
```

Build, Install and Run the Application

We must make a few changes to our application files based on the default naming created by the DSP Server Wizard.

11. Move back to the application directory.

`cd ../app` to move back to the lab12a_build_server/app directory

12. Edit and/or verify that the XDCPATH in our `makefile_profile.mak` includes the path to our server..

```
gedit makefile_profile.mak &
```

Did XDCPATH include the path to our server? ______________________________

What is the path to the server? ______________________________

13. Build and install the application.

```
make debug install
```

Remember, if you have reset the board since running the lab 11 application you will need to re-run the `loadmodules.sh` script.

14. Execute the `app_debug.xv5T` application. Press `ctrl-c` to exit the application.

iUniversal Lab Exercise

Lab Topics

Lab 13a – Creating a Universal Algo

Running the GenCodecPkg wizard

1. Start CCSv4.

2. Invoke *GenCodecPkg* wizard.

Try this option

Option 1: From CCSv4

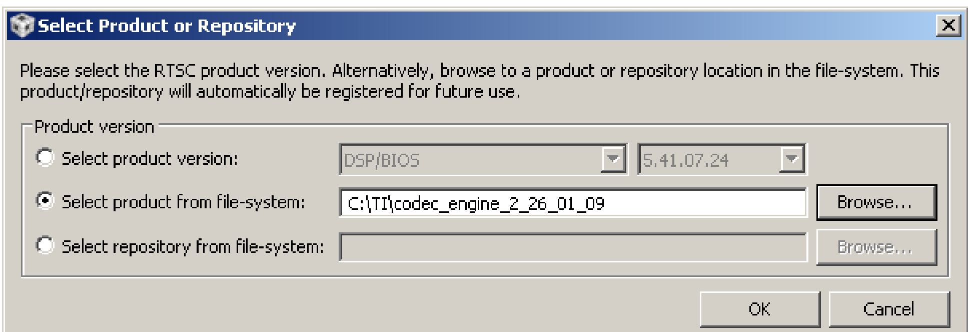
Add your Codec Engine installation directory to the "Tool Discovery Path". Go to

`Window | Preferences | CCS | RTSC`

and verify the package `codec_engine_2_26_01_09` shows up in the window. If so, check the box next to it and exit the preferences.

If it's not there:

a. Add this package from the *C:/TI* directory.



b. Close and save the dialog.

c. Restart CCS. (At this point, it's the easiest way to restart workspace).

d. Then reopen the same dialog to check the checkbox for the codec engine, then close the dialog.

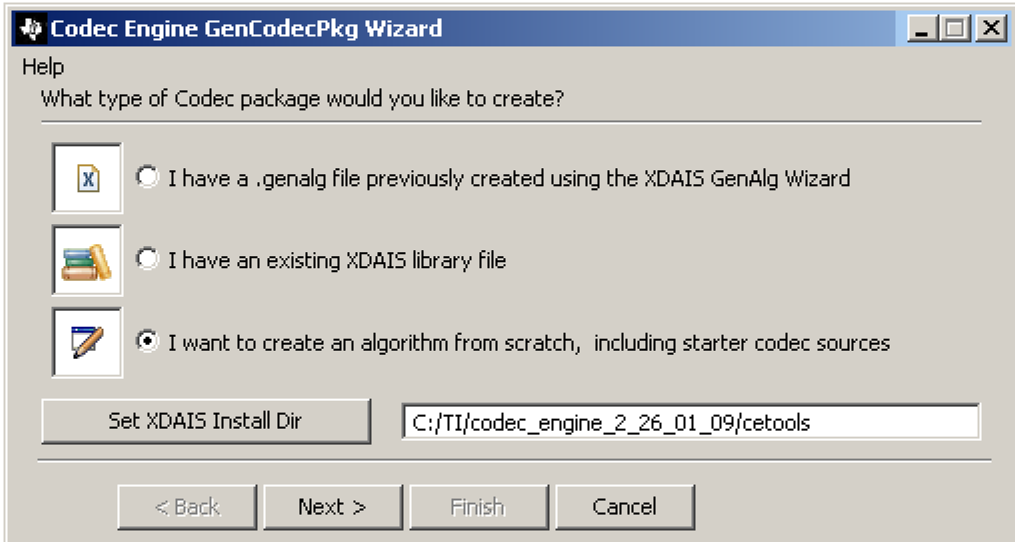
At this point, you will have a new "**Tools** | **Codec Engine Tools** | **GenCodecPkg**" menu item that will launch the wizard. Go ahead and startup the wizard.

Option 2: Create a simple makefile to invoke the wizard

```
# Paths to required dependencies
XDC_INSTALL_DIR   := C:/ti/xdctools/xdctools_3_20_03_63
CE_INSTALL_DIR    := C:/ti/codec_engine/codec_engine_2_26_01_09
XDAIS_INSTALL_DIR := $(CE_INSTALL_DIR)/cetools

# Create an XDCPATH variable
export XDCPATH = $(CE_INSTALL_DIR)/packages;$(XDAIS_INSTALL_DIR)/packages

.PHONY : gencodecpkg
gencodecpkg:
	$(XDC_INSTALL_DIR)/xs ti.sdo.ce.wizards.gencodecpkg
```

From the command-line, start the wizard with: `make gencodecpkg`

3. Generate IUNIVERSAL starterware – GenCodecPkg page 1.



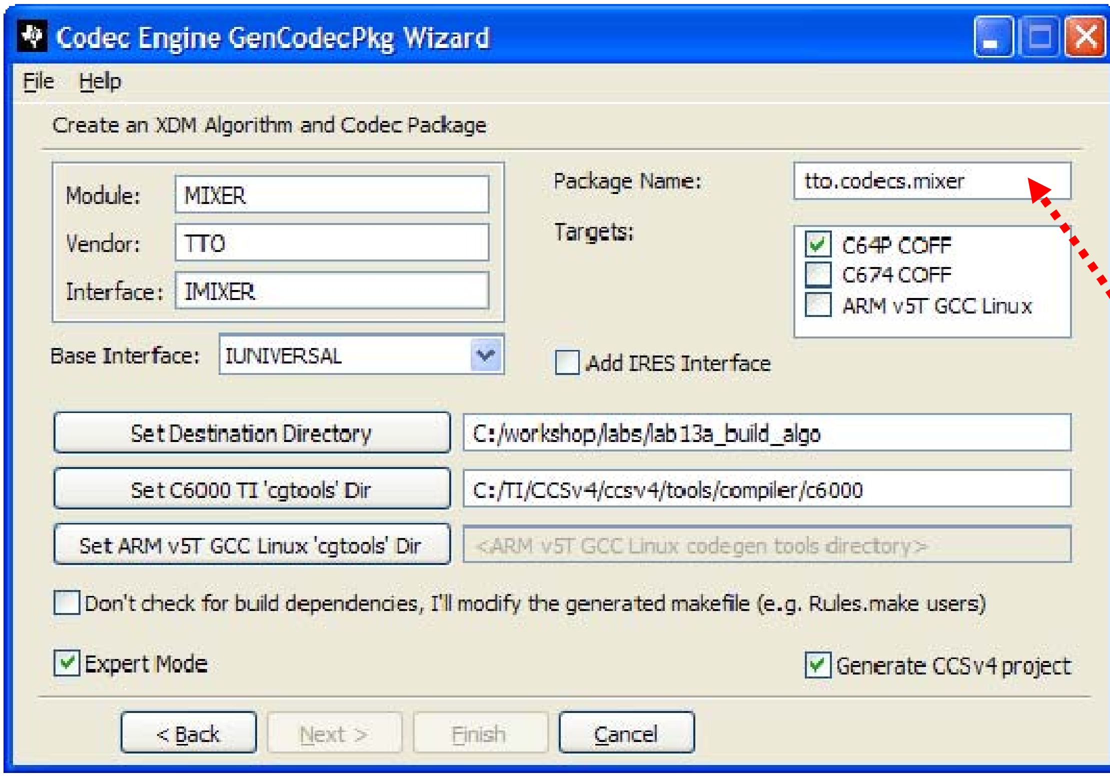
a. Click 3rd option, "I want to create an algorithm from scratch".

b. The XDAIS directory should already point to your XDAIS directory (frequently the `$(CE_INSTALL_DIR)/cetools` directory if you're not using a SDK).

c. Click Next.

> **Sidenote**
>
> When downloading the Codec Engine (stand-alone) from TI, you can choose either the standard or *lite* versions.
>
> The standard version contains an extra *cetools* directory which includes a number of additional packages which CE depends on – such as XDAIS.
>
> The SDK team has chosen to install the lite version; then they install all the other packages at the root level of the SDK directory.

4. Generate IUNIVERSAL starterware – GenCodecPkg page 2.



a. In the "**Module**" field enter the name of the algorithm you're creating: MIXER

b. In the "**Vendor**" field enter your company name: TTO

c. Under "**Base Interface**" choose: IUNIVERSAL

d. We want to build for C64x+ devices, so for "**Target**" choose: C64P COFF

e. The **Destination Directory** is where the generated files will be placed. Click the button and create (and then select) the directory shown above.

f. The **C6000 TI 'cgtools' Directory** should point to your TI DSP compiler (the root of the compiler installation, just above above the "bin" directory). When invoking the wizard from within CCSv4, this is probably already set for you.

g. Check "**Expert Mode**" and change the "**Package Name**" to match that above. (While not required, we wanted to organize our files in this way.)

h. Make sure that "**Generate CCSv4 project**" is checked.

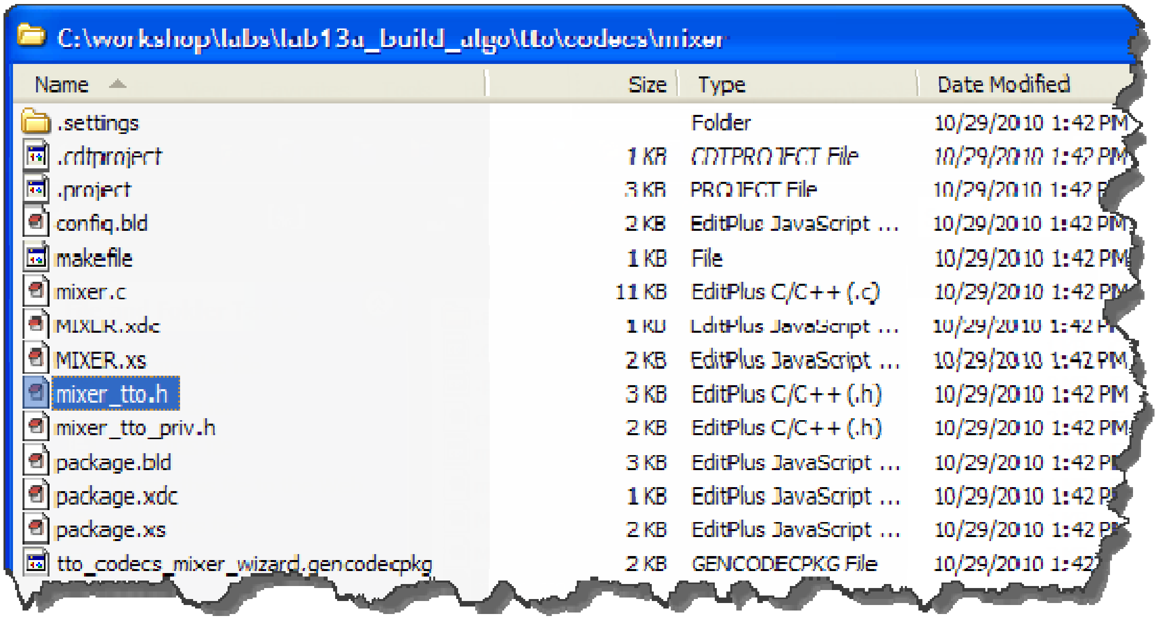
i. Click Finish to generate the starter files.

5.

Common Mistake

Make sure you change the Package Name, so that your directory paths are named like those in this lab write-up.

View wizard's generated output files.

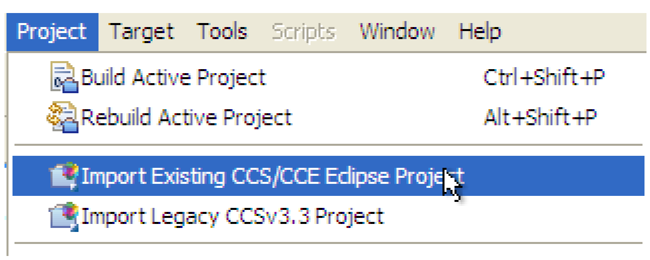


The generated files will reside at:

$\underbrace{\texttt{C:\textbackslash workshop\textbackslash labs\textbackslash lab13a\_build\_algo}}_{\textbf{< repository >}}\underbrace{\texttt{\textbackslash tto\textbackslash codecs\textbackslash mixer}}_{\textbf{< package name >}}$

Import Codec Library Project into CCSv4

6. Open CCSv4 and "Import Existing Project".



Import the codec wizard project we just created in:

```
C:\workshop\labs\lab13a_build_algo
```

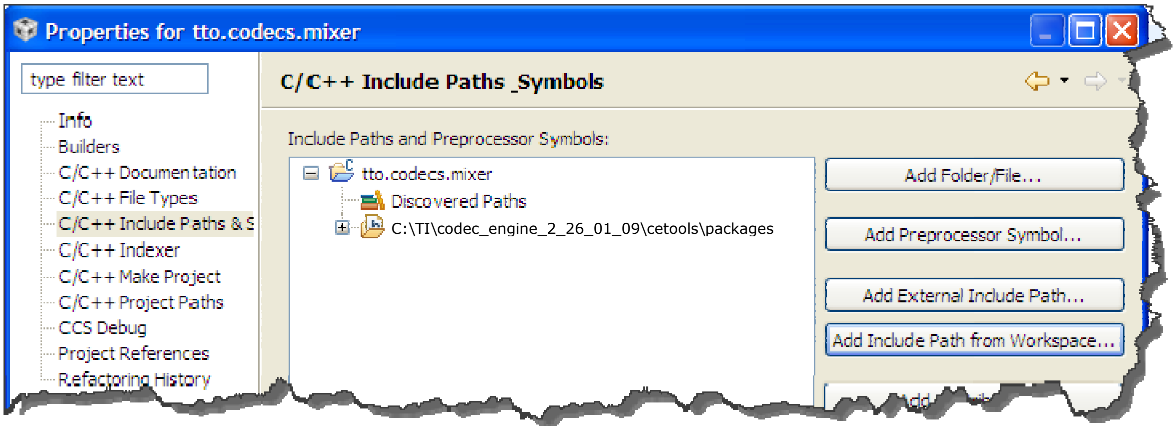
7. <u>Select</u> the project you just imported and <u>build</u> it.

Before making any changes, you should be able to build the generated package as is. It's worth verifying the generated files build correctly.

```
Select Project | Right-Click | Build Project
```

8. Setup code completion for CCSv4.

A good reason to use CCS is to take advantage of features like code completion. Later, when debugging, if you press <Ctrl> <Space> on your keyboard you can invoke the Eclipse code completion feature, which will help to finish typing the names of variables, functions, etc. You will likely want CCS to have the capability of auto-completing names/fields related to XDAIS's IALG libraries.



To set this up, do the following:

a. Right-click on your project in the "**C/C++ Projects**" window and select Properties.

b. Click on "**C/C++ Include Paths & Symbols**".

c. Click "**Add External Include Path**".

d. Click "**Browse**" and navigate to the appropriate source directory. Specifically for IALG definitions, you need to point to *<xdais>\packages*.

```
C:\TI\codec_engine_2_26_01_09\cetools\packages
```

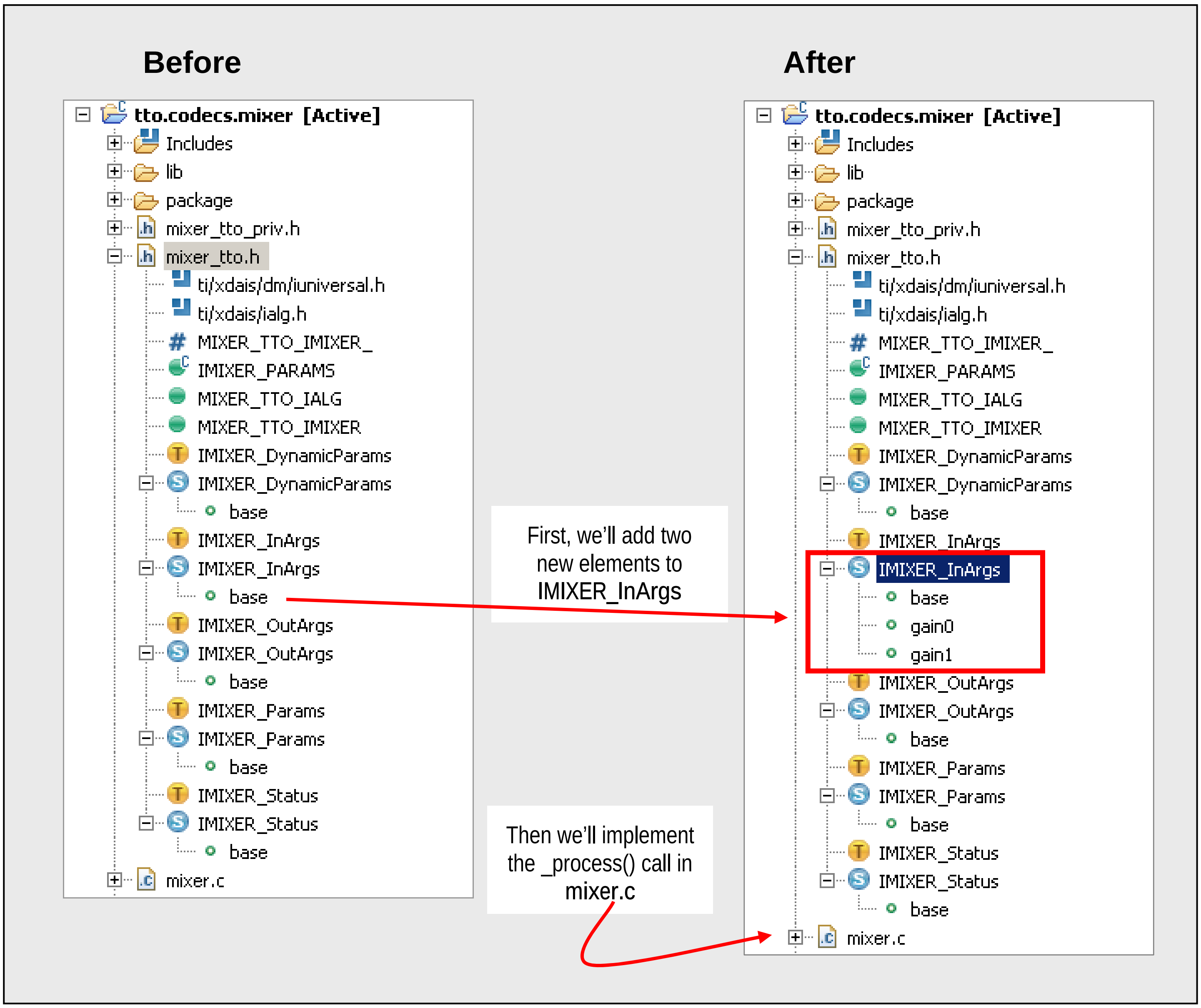
> **Note**
>
> *The XDAIS path shown in this step assumes that you have installed the full version of the Codec Engine, which includes a number of other libraries in the "cetools" directory. If you happened to install the "lite" version of the Codec Engine, then you would need to have installed the XDAIS library separately – and would want to use that path instead.*

Customizing the Code to Fit Your Algorithm

We have chosen to implement a 2-channel mixer algorithm. This algo takes two buffers, weights them each with their own gain value, then adds the buffers together.

Here is the data we will need to pass to our algo's _process() function:

Two **InBufs**
One **OutBufs**
Two Additional elements for **InArgs**:
gain0
gain1



9. Modify the algo's data structures.

Since the buffers descriptors can take a variable number of buffers, we don't really need to modify the wizard's output for them. We will have to add the gain variables to our code, though.

Please refer to the *"Getting Started with IUNIVERSAL"* wiki page – Step 4 : Customizing the Code to Fit Your Algorithm. (We provided this in PDF format in the **`lab13_starter_files`** folder.)

Some hints on what to edit when following the Getting Started with IUNIVERSAL:

mixer_tto_priv.h

e. Edits to <module>_<vendor>_priv.h:

`No changes needed…`

mixer_tto.h

f. Edits to <module>_<vendor>.h (in our case: **`mixer_tto.h`**):

– Define commands for use in the **control()** function.

`No changes needed…`

– Extend the **`I<MODULE>_Params`** as needed.

`No changes needed…`

– Extend the **`I<MODULE>_InArgs`** structure as needed.

Type in By Hand

```
typedef struct IMIXER_InArgs {
    /* This must be the first field */
    IUNIVERSAL_InArgs base;

    XDAS_Int16 gain0;
    XDAS_Int16 gain1;
```

– Extend the `I<MODULE>_OutArgs` structure as needed.

`No changes needed…`

– Extend the `I<MODULE>_DynamicParams` structure as need.

`No changes needed…`

– Extend the `I<MODULE>_Status` structure as necessary.

`No changes needed…`

mixer.c Edits to <module>.c

No changes needed

- const I<MODULE>_Params I<MODULE>_PARAMS
- <MODULE>_<VENDOR>_alloc()
- <MODULE>_<VENDOR>_free()
- <MODULE>_<VENDOR_initObj()
- <MODULE>_<VENDOR>_process()

Cut & Paste

Edit _process() call

This is where the actual algorithm goes! Add your algorithm code here.

Note, we provided the entire ***process()*** *function code for you to cut/paste to help minimize typos.*

```
/*
 *  ======== MIXER_TTO_process ========
 */

/* ARGSUSED - this line tells the TI compiler not to warn about unused args. */
    XDAS_Int32 MIXER_TTO_process(IUNIVERSAL_Handle h,
        XDM1_BufDesc *inBufs, XDM1_BufDesc *outBufs, XDM1_BufDesc *inOutBufs,
        IUNIVERSAL_InArgs *universalInArgs,
        IUNIVERSAL_OutArgs *universalOutArgs)
{
    XDAS_Int32 numInBytes, i;
    XDAS_Int16 *pIn0, *pIn1, *pOut, gain0, gain1;

    /* Local casted variables to ease operating on our extended fields */
    IMIXER_InArgs *inArgs = (IMIXER_InArgs *)universalInArgs;
    IMIXER_OutArgs *outArgs = (IMIXER_OutArgs *)universalOutArgs;

    /*
     * Note that the rest of this function will be algorithm-specific.  In
     * the initial generated implementation, this process() function simply
     * copies the first inBuf to the first outBuf.  But you should modify
     * this to suit your algorithm's needs.
     */

    /*
     * Validate arguments.  You should add your own algorithm-specific
     * parameter validation and potentially algorithm-specific return values.
     */
    if ((inArgs->base.size != sizeof(*inArgs)) ||
            (outArgs->base.size != sizeof(*outArgs))) {
        outArgs->base.extendedError = XDM_UNSUPPORTEDPARAM;

        return (IUNIVERSAL_EUNSUPPORTED);
    }

    /* validate that there's 2 inBufs and 1 outBuf */
    if ((inBufs->numBufs != 2) || (outBufs->numBufs != 1)) {
        outArgs->base.extendedError = XDM_UNSUPPORTEDPARAM;

        return (IUNIVERSAL_EFAIL);
    }

    /* validate that buffer sizes are the same */
    if (inBufs->descs[0].bufSize != inBufs->descs[1].bufSize)
                  return IUNIVERSAL_EFAIL;
    if (inBufs->descs[0].bufSize != outBufs->descs[0].bufSize)
                  return IUNIVERSAL_EFAIL;
```

✍ *Continued on next page*

```
// DO IT!
    pIn0 = (XDAS_Int16*)inBufs->descs[0].buf;
    pIn1 = (XDAS_Int16*)inBufs->descs[1].buf;
    pOut = (XDAS_Int16*)outBufs->descs[0].buf;
    gain0 = inArgs->gain0;
    gain1 = inArgs->gain1;
    numInBytes = inBufs->descs[0].bufSize;

    for(i=0; i<numInBytes/2; i++)
    {
       pOut[i] = (pIn0[i]*(XDAS_Int32)gain0 + pIn1[i]*(XDAS_Int32)gain1) >> 15;
    }

    /* report how we accessed the input buffers */
    inBufs->descs[0].accessMask = 0;
    XDM_SETACCESSMODE_READ(inBufs->descs[0].accessMask);
    inBufs->descs[1].accessMask = 0;
    XDM_SETACCESSMODE_READ(inBufs->descs[1].accessMask);

    /* report how we accessed the output buffer */
    outBufs->descs[0].accessMask = 0;
    XDM_SETACCESSMODE_WRITE(outBufs->descs[0].accessMask);

    /*
     * Fill out the rest of the outArgs struct, including any extended
     * outArgs your algorithm has defined.
     */
    outArgs->base.extendedError = 0;

    return (IUNIVERSAL_EOK);
}
```

IMPORTANT

For the purpose of cache coherence it's important to tell the framework about how the CPU has accessed the buffers.

You must tell the application if you have read from or written to any of the buffers. This is achieved by using the `accessMask` fields associated with each and every buffer.

– `<MODULE>_<VENDOR>_control()`

   - By default this function already supports the required command XDM_GETVERSION.

   - There is an `#ifdef 0` that you can get rid of to add handling for any commands that you defined in <module>_<vendor>.h, (i.e. `mixer_tto.h`).

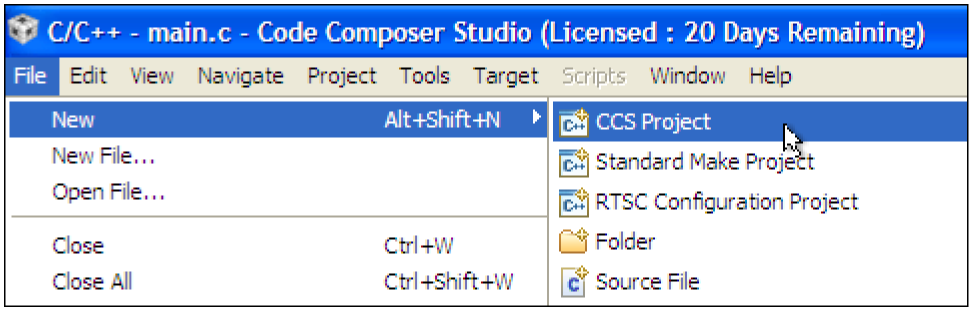
10. Build your modified algorithm to verify it's free from C language errors.

Use CCSv4 to build the algorithm. Keep debugging the algorithm until it builds correctly.

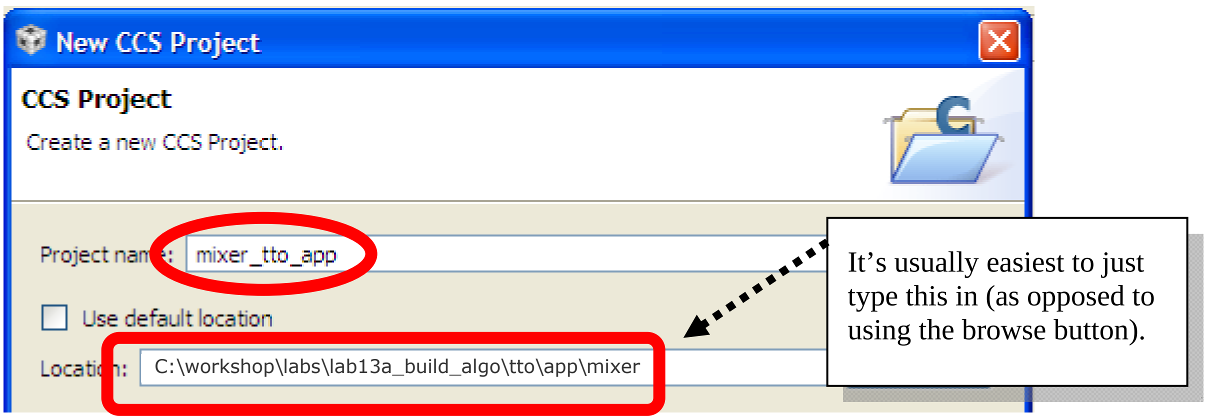
In the next part, we will test the algorithm to verify it is logically correct. For now, we just want to make sure we have not introduced any C errors.

Create a CCS Project for our Test Algorithm

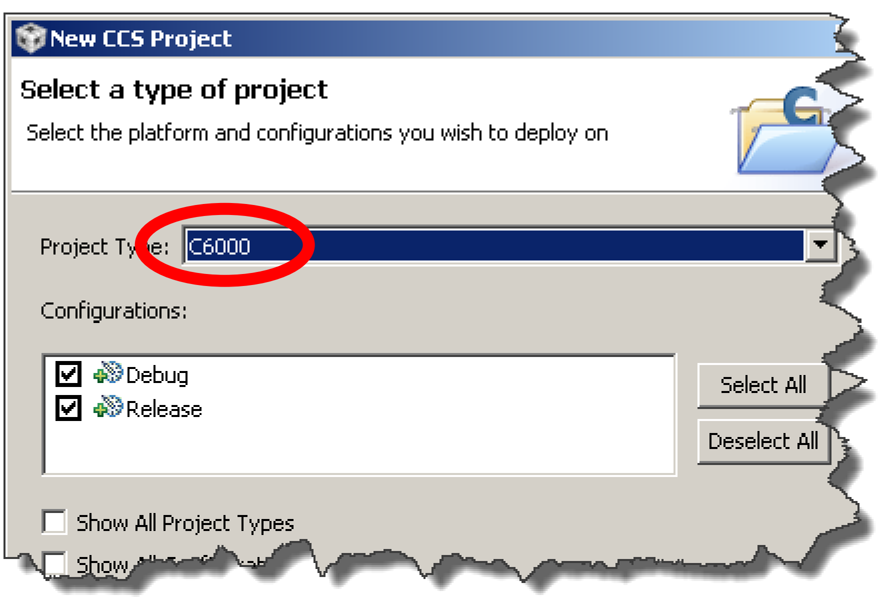
11. Create a new CCS Project.



12. Name and locate the project as shown below.

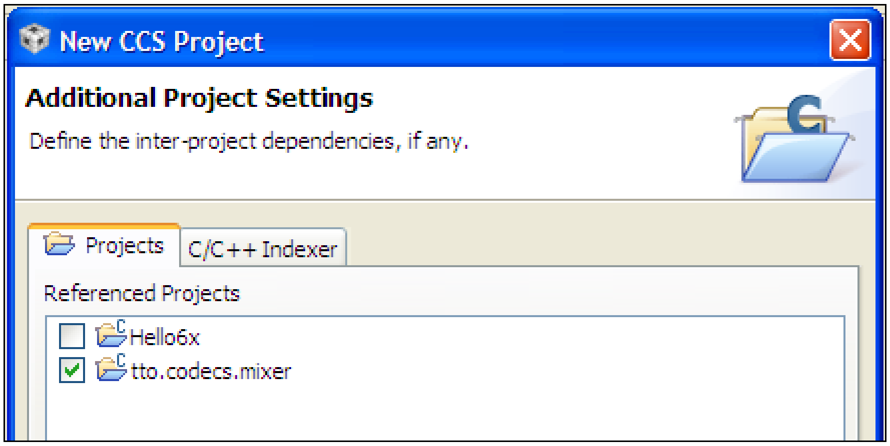


13. Choose the C6000 compiler target.

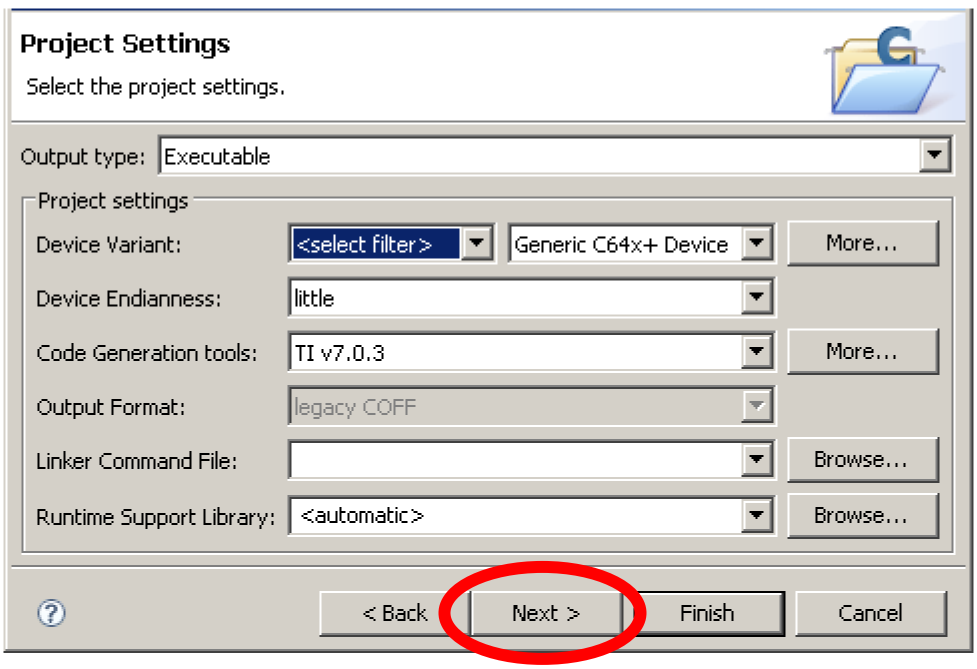


14. Select your codec project as a *dependent* project.

This makes it easy to debug code from both projects at the same time.



15. Set the project settings for running on the C64x+ DSP.



Choose:

g. Generic C64x+ Device

h. Little endian

i. The default TI DSP compiler

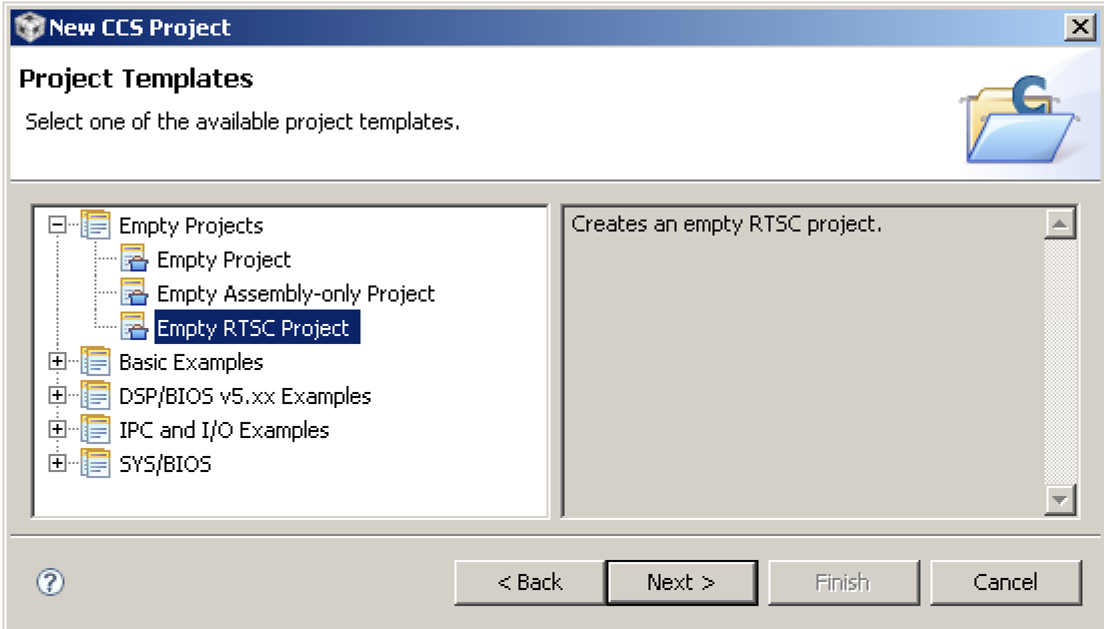
and then hit **Next**.

Avoid Common Mistake

Make sure you click NEXT!

16. Create an *Empty RTSC Project*.

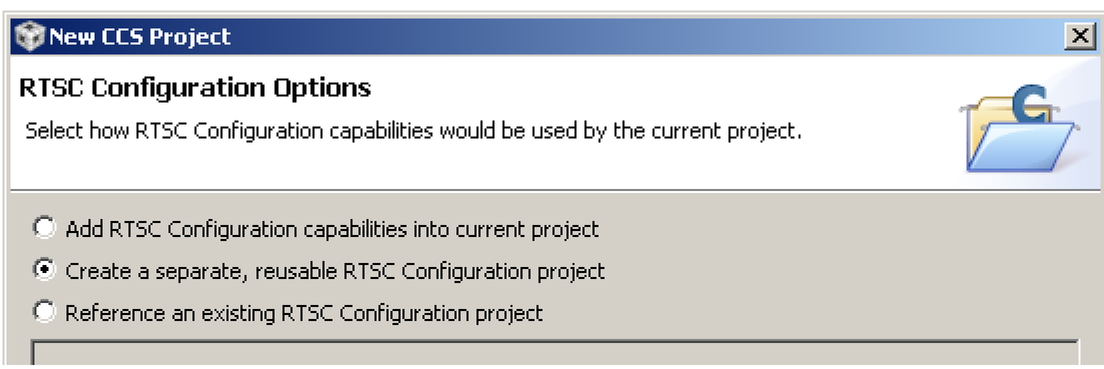
We want to create a RTSC project so we can make use of our new RTSC packaged algorithm. So, choose *Empty RTSC Project* and click *Next*.



17. Create a "separate, reusable RTSC Configuration project".

Based on the "**Empty RTSC Project**" template from the previous dialog box, CCS will ask if you already have a RTSC config project or want to create a new one. In our case, we'll be creating a new one.

While we could just add the RTSC config info to our current project (item 1 below), we have chosen to create a stand-alone RTSC config project.



When you select this option, CCS will create a new CCS project (of type RTSC Config Project). This means you'll actually have 3 CCS projects now:

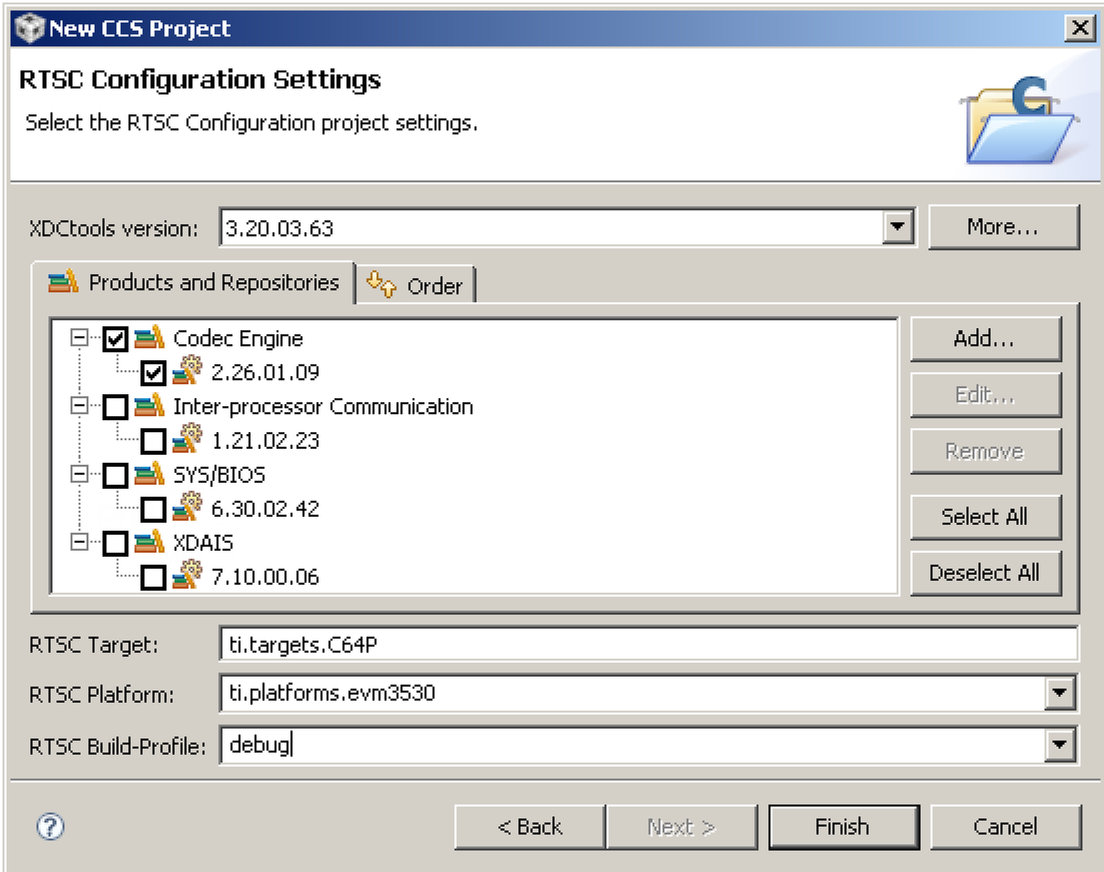
j. Algorithm

k. Test application

l. RTSC configuration

The RTSC configuration project is the way CCS can run Configuro. This is similar to adding the Configuro step to our makefile in previous labs.

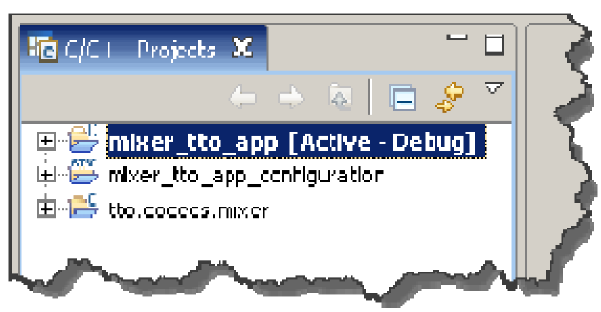
18. **Choose the CCSv4 installed packages that will be included with the project.**



Fill-out the dialog as shown above, then click Finish.

Note: we'll add more packages in our `codec.cfg` *file in an upcoming step.*

19. **You should now see three projects:**



Setup the new RTSC Config Project

20. Add the CE and BIOS configuration files to your new RTSC config project.

The Configuro allows us to consume RTSC packages that are specified in a `.cfg` file.

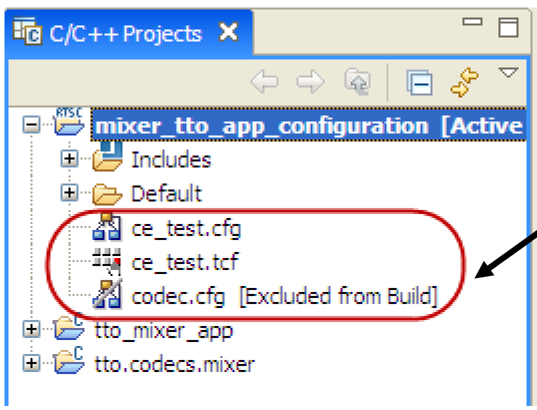
In CCS, we will need to add our .cfg file to the RTSC configuration project. In this way, it will end up doing the same thing as our Configuro command performed in our standard make file.

You can right-click on the *mixer_tto_app_configuration* project, select *Add File to Project…*, and add the following files from the Lab13a starter files:

```
ce_test.cfg
cd_test.tcf      from C:\workshop\labs\lab13a_starter_files\config
codec.cfg
```

At the end of this step, your project should look something like:

Important !!!



Make sure that "codec.cfg" is the file which is excluded from the build.

You can do this by right-clicking on the individual files and selecting whether to include/exclude it from the build process.

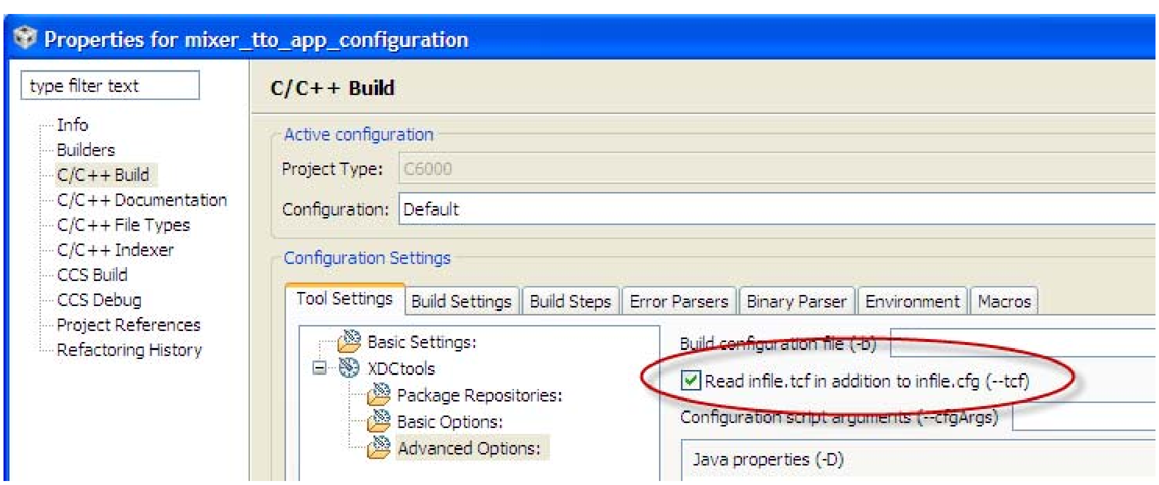
The reason we're excluding codec.cfg is that it's actually included (similar to #include) into ce_test.cfg. The XDC tools require the .cfg and .tcf files to be named the same.

21. Change [Excluded from Build] – as shown above.

So many folks miss this little item, we decided to call it out specifically.

22. Tell the config project to read both `.tcf` and `.cfg` files.

Right-click and open the properties of the mixer_tto_app_configuration project and check the box under the: C/C++ Build | Advanced Options:

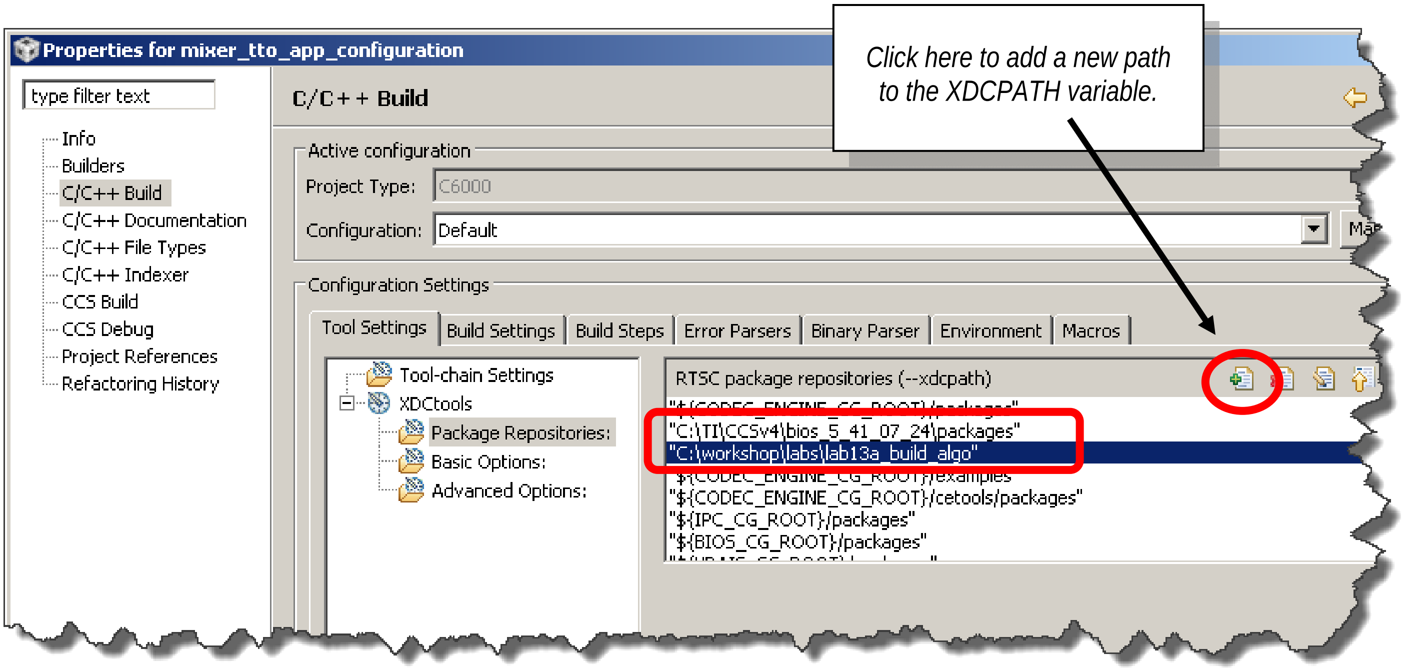


23. While you have the properties open, we need to add to the search path (i.e. XDCPATH).

Our config project needs two more paths added to its XDCPATH search list:

```
C:\TI\CCSv4\bios_5_41_07_24\packages
C:\workshop\labs\lab13a_build_algo
```

Without adding the lab13a path, the config project wouldn't be able to find the algorithm we've just created.



Setup the Algorithm Test Project

The test project allows us to run and test our algorithm. Without this project, we wouldn't know if the algorithm really gives the correct answer. We know it builds, but we also want to know it works logically, too.

24. Add the test program – `main.c`.

We have written a simple test application that will use the UNIVERSAL VISA calls to test your algorithm.

```
Right-click mix_tto_app and select Add Files to Project…
```

Add the following file to your project:

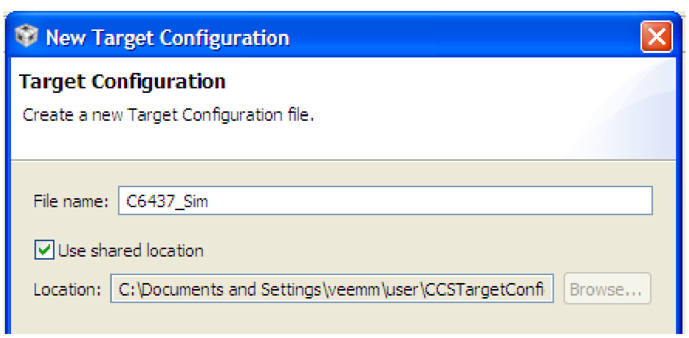
```
C:\workshop\labs\lab13a_starter_files
```

25. Create a Target Configuration File for the C64x+ Simulator.

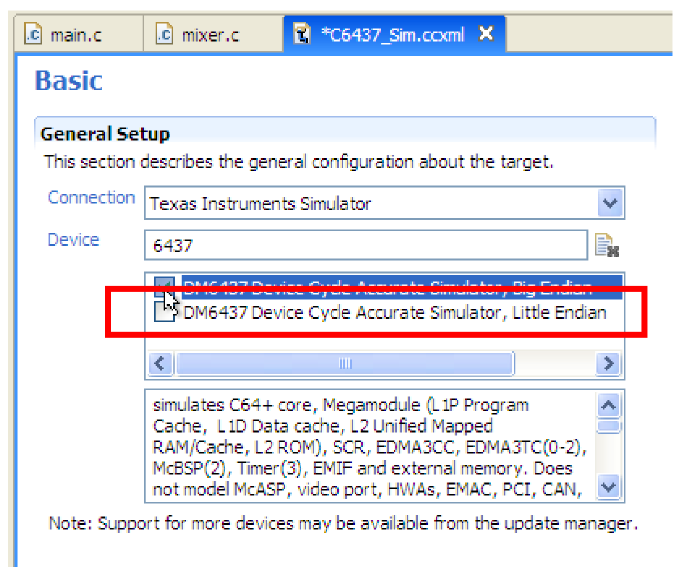
CCSv4 uses *Target Configuration Files* to specify which target you want to run (i.e. debug) your code on. (In CCSv3.3, you were required to run CCS Setup which set the target for all projects withing CCS. CCSv4 allows each project to define a different target it will run on, which is very nice.)

```
New  |  Target Configuration Files
```

Name the file something like C6437_Sim and let's just choose to use the shared location. (The shared location just means that every project in the workspace can use this Target Config file).



Pick the *DM6437 Device Cycle Accurate Simulator*, ***Little Endian***.



26. Build the test application.

You shouldn't have any errors to fix … our fingers are crossed.

Run and Debug Algorithm

27. Start a debug session.

Click the debug icon to start a debug session and download the program to the simulator. Also, CCSv4 will change to a debug perspective (i.e. window layout).

28. Set breakpoints on the 4 VISA functions in our `main()` function.

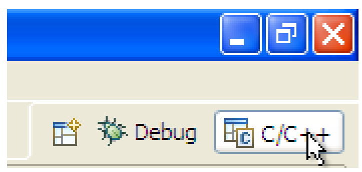
```
MIXER_create()
MIXER_process()
MIXER_control()
UNIVERSAL_delete()
```

29. Set breakpoints in the algorithm, too.

One of the big conveniences of debugging our test application – along with the algorithm library package as a dependent project – is the ability to set breakpoints right in the algo itself.

```
Click on the C/C++ perspective button
```

to go back into the project/editing window layout.

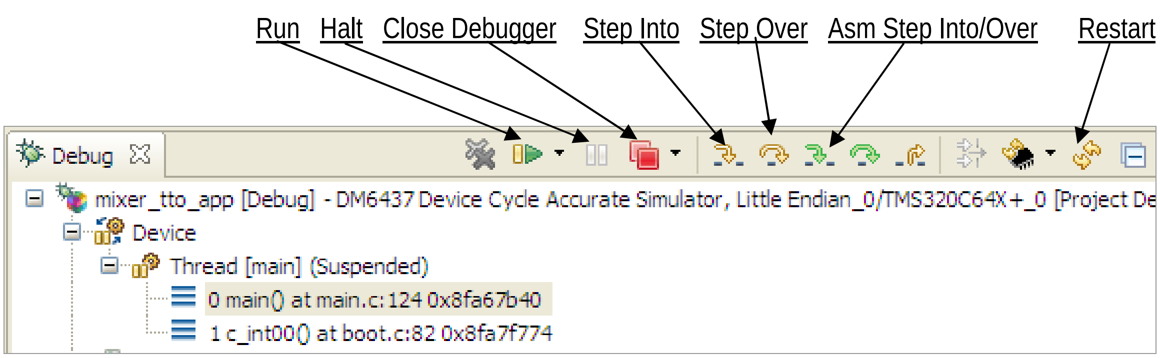


Navigate to the **`mixer.c`** file (in the `tto.codecs.mixer` project) and set a breakpoint inside the **`MIXER_TTO_process()`** function. For example, maybe set a breakpoint on the line:

```
if ((inArgs->base.size != sizeof(*inArgs)) ||
```

30. Return to the Debug perspective and run/step thru the code.

Some hints when running the debugger.



Note, you may see an error after the MIXER_control() call returns. At this point, the string return is not working properly. Sorry, we'll figure it out later … or you can take on the challenge.

31. When you're finished testing the code, close the debugger.

You can also close CCSv4 now, too — if you want to.

Test Application Code

```
//============================================================================
//
//    FILE NAME : main.c
//
//    ALGORITHM : MIXER
//
//    VENDOR    : TTO
//
//    TARGET DSP: C64x+
//
//    PURPOSE   : This file contains code to test the MIXER algorithm.
//
//    Nextxen Algorithm Wizard Version 1.00.00 Auto-Generated Component
//
//    Creation Date: Mon - 17 January 2011
//    Creation Time: 12:51 AM
//
//============================================================================

#include <xdc/std.h>
// which includes: stdarg.h, stddef.h

#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/osal/Memory.h>             // Contiguous memory alloc functions
#include <ti/sdo/ce/universal/universal.h>
// which includes: iuniversal.h, xdm.h, ialg.h, xdas.h, Engine.h, visa.h, skel.h

#include <stdint.h>                            // used for definitions like uint32_t, ...
#include <string.h>                            // used for memset/memcpy commands

// Note1: Make sure you have added this algorithms repository as a -i compiler option;
that is, make sure the path that contains the folder "tto" is provided as a -i include
path. The compiler will concatenate the -i path, along with the specified in the <> to
find the actual header file.
// Note2: The 'mixer_tto.h' header also includes: imyalg.h

#include <tto/codecs/mixer/mixer_tto.h>

//============================================================================
//
//    Prototypes
//
//============================================================================
int main ( void );
void setup_IMIXER_buffers ( void );
void error_check ( Uint32 event, XDAS_Int32 val );

//============================================================================
//
//    Global/Static Variables
//
//    Note: We chose to make most of the variables and arrays 'static local'
//          to make debugging easier for our simple codec engine test example.
//          By declaring them this way, they remain in scope during the entire
//          program.
//
//============================================================================
#define PROGRAM_NAME        "universal_test"
#define ENGINE_NAME         "myEngine"
#define ALGO_NAME           "mixer"

static String               sProgName   = PROGRAM_NAME;
static String               sEngineName = ENGINE_NAME;
static String               sAlgoName   = ALGO_NAME;

static Engine_Handle        hEngine     = NULL;
static UNIVERSAL_Handle     hUniversal  = NULL;
```

```
static IMIXER_Params         myParams;

#define IN_BUFFER_SIZE 20
#define OUT_BUFFER_SIZE 20
#define INOUT_BUFFER_SIZE 16
#define STATUS_BUFFER_SIZE 16

static XDAS_Int8              *in0;              // [IN_BUFFER_SIZE];
static XDAS_Int8              *in1;              // [IN_BUFFER_SIZE];
static XDAS_Int8              *out0;             // [OUT_BUFFER_SIZE];
static XDAS_Int8              *status0;          // [STATUS_BUFFER_SIZE];

static XDM1_BufDesc           myInBufs;
static XDM1_BufDesc           myOutBufs;
static XDM1_BufDesc           myInOutBufs;
static IMIXER_InArgs          myInArgs;
static IMIXER_OutArgs         myOutArgs;
static IMIXER_DynamicParams   myDynParams;
static IMIXER_Status          myStatus;

#define RETURN_ERROR 0
#define RETURN_SUCCESS 1

static unsigned int          funcReturn  = RETURN_SUCCESS;

static XDAS_Int32            rStatus     = 0;

// ==============================================================================
// MIXER_ Function Macros
//
// The following three macros make it easier to call the algorithm's create, process,
// and control methods.  They provide recasting of the functions and arguments from
// the MIXER algorithm, to the UNIVERSAL API.  This is needed since the Codec Engine
// framework implements the common API, which provides portability and ease-of-use.
//
#define MIXER_create(hEngine, sAlgoName, Params)
        UNIVERSAL_create(hEngine, sAlgoName, (IUNIVERSAL_Params *)&Params)
#define MIXER_process(hUniversal, InBufs, OutBufs, InOutBufs, InArgs, OutArgs)
        UNIVERSAL_process(hUniversal,
                          &InBufs,
                          &OutBufs,
                          &InOutBufs,
                          (IUNIVERSAL_InArgs *)&InArgs,
                          (IUNIVERSAL_OutArgs *)&OutArgs )
#define MIXER_control(hEngine, eCmdId, DynParams, Status)
        UNIVERSAL_control(hEngine,
                          eCmdId,
                          (UNIVERSAL_DynamicParams *)&DynParams,
                          (UNIVERSAL_Status *)&Status )


//===============================================================================
//
//    Functions
//
//===============================================================================

int main(void)
{

//  ====  Open the Codec Engine ==============================================

    CERuntime_init();

    hEngine = Engine_open(sEngineName, NULL, NULL);
    error_check(TEST_ENGINE_OPEN,(XDAS_Int32) hEngine);
```

```
//  ====  Create an Algo Instance ============================================

    // Initialize the params used for the create call
    myParams.base.size = sizeof( IMIXER_Params );

    //The MIXER_create() function creates an instance of our algorithm; you
    // can call the generic UNIVERSAL_create() function, but you would need to
    // correctly cast the parameters. The iMIXER.h file defines macros which
    // simplify the _create, _process, and _control function calls.
    hUniversal = MIXER_create(hEngine, sAlgoName, myParams);

    error_check(TEST_ALGO_CREATE, (XDAS_Int32) hUniversal);


//  ====  Run the Algorithm ==================================================

    setup_IMIXER_buffers();

    // Default values were applied; please change if you want to select other values.
    myInArgs.base.size = sizeof(IMIXER_InArgs);
    myInArgs.gain0 = 0x3fff;
    myInArgs.gain1 = 0x3fff;

    //IMIXER_OutArgs was not extended, so no additional values must be set in myOutArgs.
    myOutArgs.base.size = sizeof(IMIXER_OutArgs);

    rStatus = MIXER_process(hUniversal, myInBufs, myOutBufs, myInOutBufs, myInArgs, myOutArgs);

    error_check(TEST_ALGO_PROCESS,rStatus);


//  ====  Call Algo control function ==========================================

    //IMIXER_DynamicParams was not extended, so no additional values must be set
    myDynParams.base.size = sizeof(IMIXER_DynamicParams);
    myStatus.base.size = sizeof(IMIXER_Status);

    rStatus = MIXER_control(hUniversal, XDM_GETVERSION, myDynParams, myStatus);

    if (!rStatus)
    {
        printf("Program '%s': Algo '%s' control call succeded\n",sProgName, sAlgoName);

        printf("\tAlg version:  %s\n", (rStatus == UNIVERSAL_EOK ?
              ((char *)myStatus.base.data.descs[0].buf) : "[unknown]"));
    }
    else
    {
        fprintf(stderr, "Program '%s': ERROR: Algo '%s' control call failed;
                rStatus=0x%x\n", sProgName, sAlgoName, (unsigned int) rStatus);
    }


//  ====  Delete and Close the Algo Instance ==================================

    UNIVERSAL_delete(hUniversal);

    Engine_close (hEngine);

//  ====  Return from Main Function ===========================================

#ifdef _DEBUG_
    printf("Program '%s': Function main() now exiting.\n", sProgName);
#endif

    //while(1);

    return(funcReturn);
}

// ============================================================================
```

```
// Buffer setup
// ==========================================================================

void setup_IMIXER_buffers(void)
{

    //  ====  ALLOCATE BUFFERS  ===============================================
    //
    //  - Buffers are allocated with the Codec Engine's contigAlloc function
    //  - On ARM, this fxn alloc's memory from the CMEM driver, which is req'd
    //    when passing data to the DSP. A contiguous allocation is made when
    //    run on the DSP, but this maps to a simple MEM allocation.
    //  - This prevents a failure on architectures like OMAP3530 which provide
    //    an MMU in the DSP's memory path.

    in0     = Memory_contigAlloc( IN_BUFFER_SIZE, 8 );
    in1     = Memory_contigAlloc( IN_BUFFER_SIZE, 8 );
    out0    = Memory_contigAlloc( OUT_BUFFER_SIZE, 8 );
    status0 = Memory_contigAlloc( STATUS_BUFFER_SIZE, 8 );

    //  ====  INITIALIZE BUFFERS  =============================================
    //
    //  - We chose to use a simple data set for testing our algorithm.
    //  - In "real" life you would want to enhance this test program with test
    //    data that validates your algorithm.

    memset( in0,    0xA, IN_BUFFER_SIZE    );
    memset( in1,    0xB, IN_BUFFER_SIZE    );
    memset( out0,   0x0, OUT_BUFFER_SIZE   );

    //  === Setup buffer descriptors for calls used in the MIXER_process()
    //      and MIXER_control() functions

    myInBufs.numBufs                 = 2;
    myInBufs.descs[0].bufSize = IN_BUFFER_SIZE;
    myInBufs.descs[0].buf     = (XDAS_Int8 *) in0;
    myInBufs.descs[1].bufSize = IN_BUFFER_SIZE;
    myInBufs.descs[1].buf     = (XDAS_Int8 *) in1;

    myOutBufs.numBufs                = 1;
    myOutBufs.descs[0].bufSize       = OUT_BUFFER_SIZE;
    myOutBufs.descs[0].buf           = (XDAS_Int8 *) out0;

    myInOutBufs.numBufs              = 0;

    myStatus.base.data.numBufs              = 1;
    myStatus.base.data.descs[0].bufSize     = STATUS_BUFFER_SIZE;
    myStatus.base.data.descs[0].buf         = (XDAS_Int8 *) status0;


}
```

Lab 13b - Creating a "server" for your algorithm

Now that you've proven your algorithm works on a single-CPU DSP system, we can wrap our algorithm into a server and call it from Linux on the ARM.

Copy the necessary files into the VM shared folder

1. **Create the folder `lab13b_run_algo` in `vm_images\shared`.**

   Create the following directory.

   ```
   C:\vm_images\shared\lab13b_run_algo
   ```

2. **Create the `app` folder in `lab13b_run_algo` and copy the `main.c` file.**

   Create the `app` folder, then copy the file **`main.c`** from your CCS application project..

   ```
   C:\vm_images\shared\lab13b_run_algo\app
   ```

   ```
   Copy main.c into this folder
   ```

3. **Create the server folder .**

   ```
   C:\vm_images\shared\lab13b_run_algo\server
   ```

4. **Copy the algorithm into the `lab13b_run_algo` folder.**

   Copy the whole algorithm (`tto.codecs.mixer`) directory to the `Lab13b_run_algo` folder.

   ```
   C:\vm_images\shared\Lab13b_run_algo\tto\codecs\mixer
   ```

File management in Linux

5. **Copy the entire `lab13b_run_algo` folder into `'labs'`.**

   Open up your VMware Ubuntu Linux and copy the entire `lab13b_run_algo` folder from the VMware shared folder into your `/home/user/labs` directory.

DM6446 Labs:

You need to do the same edits as in steps 6-8, but yours will look a little different than what is shown here.

Also your files would be in the "workshops" folder, not the "labs" folder.

6. **Copy the makefile from `Lab12a/server` and edit XDCPATH.**

   ```
   Copy the makefile from your lab12a server folder into the
   lab13b_build_algo server folder
   ```

   Verify that the your current lab path is included in the XDCPATH correctly.

7. **Copy the makefiles from `lab12a_build_server/app` to `lab13b_build_algo/app` and edit them.**

   After copying the two makefiles: (1) verify that your new codec package is on the XDCPATH; and, (2) remove the OSD file from the `INSTALL_OSD_IMAGE` variable.

   INSTALL_OSD_IMAGE := ~~../osdfiles/ti_rgb24_640x80.bmp~~

8. Copy the `app_cfg.cfg` from `lab12a_build_server/app` to `Lab13b_run_algo/app`.

Again, we need to only make one small edit. Change the engine name from "*encodedecode*" to "*myEngine*" – which is the name used in our `main.c` file.

Make the DSP Server

9. Switch to the `Lab13b_run_algo/server` folder and run the DSP Server using the wizard.

Follow the steps from *lab12a_build_server* to create a new server using the "mixer" codec.

Use the same platform and server package name as we've done in the past, but remember to use the repo location: `/home/user/labs/lab13a`

10. Build the server.

Build and Test the app and algo

11. Switch to the `lab13b_build_algo/app` directory and build/install the test application.

```
make debug install
```

12. Switch over to Tera Term and run the application.

If you've reset the EVM, make sure you run `loadmodules.sh` before running the application. You should see a *printf* statement output as the program completes each VISA call. (You can review the *main.c* code to find these *printf's* in the test app – or add more if you'd like to trace things more closely..)