Linux Embedded System Design Workshop

Designing with Texas Instruments ARM and ARM+DSP Systems

Student Guide

Workshop Student Notes
Revision 2.10
August 2010

Technical Training
Notice

Creation of derivative works unless agreed to in writing by the copyright owner is forbidden. No portion of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright holder.

Texas Instruments reserves the right to update this Guide to reflect the most current product information for the spectrum of users. If there are any differences between this Guide and a technical reference manual, references should always be made to the most current reference manual. Information contained in this publication is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

Copyright © 2006 - 2010 by Texas Instruments Incorporated.
All rights reserved.

Training Technical Organization
Texas Instruments, Incorporated
7834 Churchill Way, MS 3984
Dallas, TX  75251-1903
(972) 917-3889

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
Welcome Topics

Workshop Goals................................................................................................................. 0-4
Where To Get Additional Information/Training? ............................................................... 0-6
Workshop Outline .................................................................................................................. 0-7
About You...................................................................................................................... 0-8
Administrative Topics ....................................................................................................... 0-8
Workshop Goals

Workshop Goal

“Learn how to put together a ARM-based Linux embedded system.”

◆ Given that you already know:
  • How to program in C
  • Basics of Linux
  • Understand basic embedded system concepts

◆ Provided with:
  • Linux Distribution for ARM
  • TI foundation software, incl: Codec Engine, Codecs (i.e. Algo's)
  • Tools: DVEVM, SDK/DVSDK

◆ In this workshop, you'll learn the essential skills required to put together a Linux ARM or ARM+DSP based system

Workshop Prerequisites

Pre-requisite Skills

◆ Required
  • Some knowledge of C programming
  • Basic Linux skills (i.e. shell commands, etc.)

◆ Recommended
  • Embedded system basics (memory map, linking, etc.)
  • Basic Linux programming (processes, threads, etc.)

◆ Nice to have
  • C6x DSP programming
  • Understanding of Linux device drivers
  • Video Application/System Knowledge

Not Required
  • No H/W design experience required
Focus of Linux/ARM Workshop

This workshop focuses on applications which plan to use TI’s ARM or ARM+DSP software model:

- ARM running embedded Linux
- DSP running DSP/BIOS
- Signal processing (and IPC) via Codec Engine (VISA API)
- Signal Processing Layer (CODECs and algorithms) built using xDM/xDAIS API
- Building programs with GNU Make (gMake) and TI's Real-Time Software XDC tools

What Will You Accomplish?
When you leave the workshop, you should be able to...

- Describe the basic silicon features and options of the TI high-performance ARM and ARM+DSP processors
- Draw a diagram describing the building blocks used in TI's ARM+DSP foundation software
- Build a Linux application which uses:
  - The audio and video Linux drivers provided in the Linux distribution
  - Invoke local (ARM-based) and remote (DSP-based) signal processing algorithms via Codec Engine (VISA API)
- Create the building-blocks used by the Codec Engine:
  - Use gMake & Configuro to build a signal processing Engine and DSP Server
- Algorithms/Codecs:
  - Describe the xDM/xDAIS API’s used to access algorithms
  - Write an algorithm following the XDM API
- Tools:
  - Setup the DVEVM hardware tool by configuring various U-Boot parameters
Where To Get Additional Information/Training?

Why Don’t We Cover Everything?
- In 4 days, it is impossible to cover everything. However, we do cover about an equivalent of a college semester course on the DM644x.
- We provide the following lists as a starting place for additional information.

Where can I get additional skills? (from Ti)

Texas Instruments Curriculum

<table>
<thead>
<tr>
<th>Topic</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Linux based Systems</td>
<td>ARM or ARM+DSP processors</td>
</tr>
<tr>
<td>Building BIOS based Systems</td>
<td>DSP processors</td>
</tr>
<tr>
<td>Developing Algo’s for C6x DSP’s</td>
<td>Are you writing/optimizing algorithms for latest C64x+ or C674x DSP’s CPUs</td>
</tr>
</tbody>
</table>

Online Resources:
- OMAP / Sitara / DaVinci Wiki http://processors.wiki.ti.com
- TI E2E Community (videos, forums, blogs) http://e2e.ti.com
- This workshop presentation & exercises http://processors.wiki.ti.com/index.php/OMAP™/DaVinci™_System_Integration_using_Linux_Workshop

Where can I get additional skills? (Non-Ti)

Non-Ti Curriculum

<table>
<thead>
<tr>
<th>Topic</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>“Linux For Dummies”, by Dee-Ann LeBlanc</td>
</tr>
<tr>
<td>Embedded Linux</td>
<td>Building Embedded Linux Systems”, by Karim Yaghmour</td>
</tr>
<tr>
<td></td>
<td>“Video Demystified”, Keith Jack</td>
</tr>
<tr>
<td></td>
<td>“Video Compression Demystified”, Peter Symes</td>
</tr>
</tbody>
</table>
Workshop Outline

### Linux Embedded System Design Workshop

**Introduction**
- 0. Welcome
- 1. Device Families Overview
- 2. TI Foundation Software
- 3. Introduction to Linux/U-Boot
- 4. Tools Overview

**Application Coding**
- 5. Building Programs with gMake
- 6. Device Driver Introduction
- 7. Video Drivers: V4L2 and FBdev
- 8. Multi-Threaded Systems

**Using the Codec Engine**
- 9. Local Codecs: Given an Engine
- 10. Local Codecs: Building an Engine
- 11. Remote Codecs: Given a DSP Server
- 12. Remote Codecs: Building a DSP Server

**Algorithms**
- 13. xDAIS and xDM Authoring
- 14. (Optional) Using DMA in Algorithms
- 15. (Optional) Intro to DSPLink

---

**Lab Exercises**

*Introduction*
- 3. Configure U-Boot and boot the DVEVM

*Application Programming*
- 5. Building programs with XDC; explore XDC pkg files
- 7. Setup an On-Screen Display 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)
- 12. Build a DSP Server (given DSP-based codecs)
  - Optional: Swap out video_copy codec for H.264 codec
  - Optional: H.264 A/V record & playback

---

Copyright © 2010 Texas Instruments. All rights reserved.
About Your Experience – A Show Of Hands

(No right or wrong answers, we just want to know where you’re coming from)

- **Who is building a Video application**
  - If not, what other types of applications

- **Experienced with Linux**
  - Linux command line (Bash)
  - Mounting drives, NFS

- **Linux C Programmer**
  - GCC, gMake
  - Linux threads (processes, pthread)

- **Previous TI DSP developer?**
  - Another TI processor
  - Competitor’s processor

- **Experience building Embedded Systems**
  - Memory maps and linking
  - Bootloading a processor

Administrative Topics

- **Name Tags**
- **Start & End Times**
- **Bathrooms**
- **Phone calls**
- **Lunch !!!**
- **Let us know if you’ll miss part of the workshop**
Introduction

In this chapter a cursory overview of Texas Instruments ARM and ARM+DSP devices by examining the CPU core, H/W Accelerator, and Peripheral options.

Along the way we will introduce a few topics that will be important as we work our way through the rest of the workshop chapters

Learning Objectives

At the conclusion of this chapter, you should be able to:

- List the major CPU cores offered by TI in their Sitara, DaVinci and C6000 families
- List the hardware accelerators offered by TI
- Describe the PRU, SCR and EDMA3 peripherals
- Describe the benefits and contraints of the MMU, pin-muxing, and ARM+DSP access to peripherals
- Choose a device based on the above critieria
Chapter Topics

TI ARM/ARM+DSP Devices ........................................................................................................ 1-1

TI Embedded Processors Portfolio .............................................................................................. 1-3

System Examples ........................................................................................................................ 1-4

What Processing Do You Need? .................................................................................................. 1-5
  ARM Core .................................................................................................................................. 1-6
  DSP Core ................................................................................................................................. 1-7
  Accelerator : 3D Graphics ........................................................................................................ 1-10
  Accelerator : Audio/Video Algorithms .................................................................................... 1-12
  Accelerators : Video Port (VPSS) / Display (DSS) ................................................................. 1-17
  Hardware Accellerator Summary ............................................................................................ 1-22

Peripherals .................................................................................................................................. 1-23
  PRU – Programmable Real-time Unit .................................................................................... 1-24
  Moving Data Around – SCR / EDMA3 .................................................................................... 1-25

Final Considerations .................................................................................................................. 1-26
  Memory Map & MMU .............................................................................................................. 1-26
  Access to Peripherals .............................................................................................................. 1-28
  Pin Muxing ............................................................................................................................ 1-29

Device Families .......................................................................................................................... 1-31

Choosing a Device : Web Tool .................................................................................................. 1-32
# TI Embedded Processors Portfolio

## TI Embedded Processors Portfolio

<table>
<thead>
<tr>
<th>Microcontrollers (MCUs)</th>
<th>ARM®-Based Processors</th>
<th>Digital Signal Processors (DSPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit ultra-low power MCUs</td>
<td>ARM Cortex™-M3 MCUs</td>
<td>High-performance DSPs</td>
</tr>
<tr>
<td>32-bit real-time MCUs</td>
<td>ARM Cortex-A8 MPUs</td>
<td>Low-power DSPs</td>
</tr>
</tbody>
</table>

### MSP430™
- Up to 25 MHz
- Flash: 1 KB to 256 KB
- Analog I/O, ADC, LCD, USB, RF
- Measurement, Sensing, General Purpose
- $0.49 to $9.00

### C2000™
- Delfino™
- Piccolo™
- 400 MHz to 300 MHz
- Flash: 16 KB to 512 KB
- PWM, ADC, CAN, SPI, IIC
- $1.00 to $20.00

### Stellaris®
- ARM Cortex™-M3
- Up to 100 MHz
- Flash: 64 KB to 256 KB
- USB, CAN, AD/DC, PWM, SPI
- Connectivity, Security, Motion Control, HMI, Industrial Automation
- $1.00 to $20.00

### Sitara™
- ARM Cortex™-A8 & ARM9
- 300 MHz to >1 GHz
- Cache, RAM, ROM
- USB, CAN, PCIe, EMAC
- Industrial computing, POS & portable data terminals
- $5.00 to $20.00

### OMAP™
- C5000™
- DaVinci™
- OMAP™
- 300 MHz to >1 GHz
- +Accelerator
- Cache, RAM, ROM
- USB, CAN, PCIe, EMAC
- Industrial computing, POS & portable data terminals
- $5.00 to $20.00

### C5000™
- Up to 300 MHz
- +Accelerator
- Up to 320KB RAM
- Up to 32GB ROM
- USB, ADC, MIPI, SPI, IIC
- Port: Telecom, audio, medical monitor & diag, Industrial
- $3.00 to $20.00

## Software & Dev. Tools

- Texas Instruments

---

*Linux Embedded System Design Workshop - TI ARM/ARM+DSP Devices*
### System Examples

<table>
<thead>
<tr>
<th>Video Output</th>
<th>Digital Audio</th>
<th>Analog Audio</th>
<th>Power Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet</td>
<td>Analog HD Component</td>
<td>Digital (DVI, HDMI)</td>
<td>TPS65020 Power Controller</td>
</tr>
<tr>
<td>TLK2226</td>
<td>OPA361</td>
<td>THS8200</td>
<td>for cameras</td>
</tr>
<tr>
<td>CVBS</td>
<td>S-Video Component</td>
<td>BT.656</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>FPC8574A</td>
<td>x’lator</td>
<td>AIC33</td>
<td>bqTINY™ Battery Charger</td>
</tr>
<tr>
<td>DDR2</td>
<td>PCA9306</td>
<td>ASP</td>
<td>Focus / Aperture motors</td>
</tr>
<tr>
<td>Flash</td>
<td>BT.1120</td>
<td>ASP</td>
<td>LCD</td>
</tr>
<tr>
<td>MSP430</td>
<td>SD/MMC</td>
<td>BT.1120</td>
<td></td>
</tr>
<tr>
<td>Power Mgmt:</td>
<td></td>
<td>USB 2.0</td>
<td>Storage SD/MMC, Etc.</td>
</tr>
<tr>
<td>- Vreg</td>
<td></td>
<td>UART</td>
<td>On-chip PWM</td>
</tr>
<tr>
<td>- SuperVisor</td>
<td></td>
<td></td>
<td>Focus / Aperture motors</td>
</tr>
<tr>
<td>Front Panel</td>
<td>Buttons &amp; LEDs</td>
<td>USB 2.0 (Expansion Port)</td>
<td>LCD</td>
</tr>
<tr>
<td>Buttons &amp; Feedback</td>
<td>GPIO or I2C</td>
<td>PCF8574A</td>
<td>TV Out NTSC/PAL</td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB 2.0 or RS232</td>
<td>TPS65560</td>
<td>Video TV Out</td>
</tr>
<tr>
<td>Power Management</td>
<td>TPS65020 Power Controller</td>
<td>for cameras</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>Storage SD/MMC</td>
<td>Etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB 2.0 or RS232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Management</td>
<td>TPS65020 Power Controller</td>
<td>for cameras</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>Storage SD/MMC</td>
<td>Etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB 2.0 or RS232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Management</td>
<td>TPS65020 Power Controller</td>
<td>for cameras</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>Storage SD/MMC</td>
<td>Etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB 2.0 or RS232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Management</td>
<td>TPS65020 Power Controller</td>
<td>for cameras</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>Storage SD/MMC</td>
<td>Etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB 2.0 or RS232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Management</td>
<td>TPS65020 Power Controller</td>
<td>for cameras</td>
<td>BQ27200 Gas Gauge</td>
</tr>
<tr>
<td>Storage SD/MMC</td>
<td>Etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Processing Do You Need? (CPU, H/W Accelerators)

What Types of Processing Do You Need?

For example, in an Audio/Video application, what needs to be done?

- ARM® CPU
  - Cortex-A8
  - ARM9
- Peripheral Subsystem
  - 3D Graphics Accelerator
  - PRU
- TI 'C6x DSP CPU
- Video Accel's
- Display Subsystem

Key System Blocks

An integrated solution that reduces System complexity, Power consumption, and Support costs

- Low Power
  - No heat sink or fan required. Ideal for end equipment that require air-tight, sealed enclosures

- ARM Core
  - High performance processors (375mhz- 1ghz) drive complex applications running on Linux, WinCE or Android systems

- Peripherals
  - Multiplicity of integrated peripheral options tailored for various wired or wireless applications – simplify your design and reduce overall costs

- Graphics Accelerator
  - Provides rich image quality, faster graphics performance and flexible image display options for advanced user interfaces

- C6x DSP Core
  - Off-load algorithmic tasks from the ARM, freeing it to perform your applications more quickly
  - Allows real-time multi-media processing expected by users of today’s end-products
  - Think of the DSP as the ultimate, programmable hardware accelerator
  - Video Accelerator – either stand-alone or combined with the DSP provide today’s meet today’s video demands with the least power req’d

- Display Subsystem
  - Off-loads tasks from the ARM, allowing development of rich “iPhone-like” user interfaces including graphic overlays and resizing without the need for an extra graphics card

- Prog. Real-time Unit (PRU)
  - Use this configurable processor block to extend peripheral count or I/F’s
  - Tailor for a proprietary interface or build a customized system control unit

NOTE
Features not available on all devices
ARM Core

TI ARM CPU Processor Cores

- Classic ARM Processors
- Embedded Cortex Processors
- Application Cortex Processors
- Cortex-A8

ARM9


TI's ARM core's supporting Linux...

ARM Processors : ARM+DSP

- Classic ARM Processors
- Embedded Cortex Processors
- Application Cortex Processors
- Cortex-A8

- ARM9 and Cortex-A8 provide the horsepower required to run high-level operating systems like: Linux, WinCE and Android
- ARM926 processor (375 – 450MHz) is the most popular and widely used processor in the market
- ARM Cortex™-A8 processor (600 MHz – 1 GHz) is ideal for high compute and graphic intense applications

**What Processing Do You Need?**

**TI ARM Core Devices**

<table>
<thead>
<tr>
<th></th>
<th>General Purpose ARM Only</th>
<th>General Purpose ARM+DSP</th>
<th>Video Oriented ARM / ARM+DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM926</td>
<td>AM1705</td>
<td>OMAP-L137</td>
<td>DM355</td>
</tr>
<tr>
<td></td>
<td>AM1707</td>
<td></td>
<td>DM365</td>
</tr>
<tr>
<td></td>
<td>AM1806</td>
<td>OMAP-L138</td>
<td>DM644x</td>
</tr>
<tr>
<td></td>
<td>AM1808</td>
<td>(also C6L13x)</td>
<td>DM6467</td>
</tr>
<tr>
<td>Cortex A8</td>
<td>OMAP3503</td>
<td></td>
<td>OMP3515</td>
</tr>
<tr>
<td></td>
<td>OMAP3515</td>
<td></td>
<td>OMP3525</td>
</tr>
<tr>
<td></td>
<td>AM3505</td>
<td></td>
<td>OMP3530</td>
</tr>
<tr>
<td></td>
<td>AM3515</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AM3703</td>
<td></td>
<td>DM3725</td>
</tr>
<tr>
<td></td>
<td>AM3715</td>
<td></td>
<td>DM3730</td>
</tr>
<tr>
<td></td>
<td>Sitara</td>
<td>“ ”</td>
<td>DaVinci</td>
</tr>
<tr>
<td></td>
<td>(AM)</td>
<td>(C6L, C6A8)</td>
<td>(DM)</td>
</tr>
</tbody>
</table>

**DSP Core**

**DSP : What Problem Are We Trying To Solve?**

Can’t process it if you don’t have it:
- Various DMA’s bring data in without using CPU MIPS*
- Excellent pointer addressing modes support high-thruput vector processing
- Support for Packed-Data Processing (SIMD) allows use of 64 and 256-bit buses even for 8/16/32-bit data

Most DSP algorithms can be expressed with MAC:

\[
Y = \sum_{i=1}^{\text{count}} \text{coeff}_i \times x_i
\]

for (i = 1; i < count; i++){
    \(Y += \text{coeff}[i] \times x[i];\) }

\[
\text{count}
\]
What Processing Do You Need?

'C6x CPU Architecture

- 'C6x Compiler excels at Natural C
- Multiplier (.M) and ALU (.L) provide up to 8 MACs/cycle (8x8 or 16x16)
- Specialized instructions accelerate intensive, non-MAC oriented calculations. Examples include:
  Video compression, Machine Vision, Reed Solomon, …
- While MMACs speed math intensive algorithms, flexibility of 8 independent functional units allows the compiler to quickly perform other types of processing
- 'C6x CPU can dispatch up to eight parallel instructions each cycle
- All 'C6x instructions are conditional allowing efficient hardware pipelining
What Processing Do You Need?

C6000 DSP Family CPU Roadmap

Fixed Point
- C6x
  - Video/Imaging Enhanced
  - EDMA2
- C62x
- C67x
- C67x+
- C671x
- C64x
  - L1 RAM/Cache
  - Compact Instr’s
  - EDMA3
- C674
  - Fixed and Floating Point
  - Lower power
  - EDMA3
  - PRU

Floating Point
- EDMA3
- PRU
- L1 RAM/Cache
- Compact Instr’s
- Exceptions
  - Supervisor/User modes
- Supervisor/User modes

Available on the most recent releases

C6000 DSP Family CPU Roadmap

- C6x Next
- C64x+
  - 1GHz
  - EDMA (v2)
  - 2x Register Set
  - SIMD Instr’s (Packed Data Proc)
- C674
  - Combined Instr Sets from C64x+/C67x+
  - Incr Floating-pt MHz
  - Lower power
  - EDMA3
  - PRU

- C621x
  - EDMA
  - L1 Cache
  - L2 Cache/RAM
  - Low Cost

- C62x
- C67x
- C671x
- C67x+
  - 1.2 GHz
  - EDMA3
  - SPLOOP
  - 32x32 Int Multiply
  - Enhanced Instr for FIR/FFT/Complex
  - DMAX (PRU)
  - 2x Register Set
  - FFT enhancements

- C6x Next
What Processing Do You Need?

C64x+ MegaModule Block Diagram

- Internal memories support:
  - Cache or RAM
  - Cache freeze
  - Memory Protection
  - Bandwidth Management
- IDMA
  - Moves between the three internal memories
  - Moves from internal memory to config registers
  - Different than iDMA API

Accelerator : 3D Graphics

Video ≠ Graphics

<table>
<thead>
<tr>
<th>Video</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Captured</td>
<td>Generated</td>
</tr>
<tr>
<td>Motion</td>
<td>Animation</td>
</tr>
<tr>
<td>*Integrated image</td>
<td>*Multi-layered image</td>
</tr>
<tr>
<td>*Base image + motion</td>
<td>*New delta images</td>
</tr>
<tr>
<td>YUV/YCbCr format</td>
<td>RGB format</td>
</tr>
<tr>
<td>Convert to RGB for display</td>
<td>Ready for display</td>
</tr>
<tr>
<td>Video codec : block-based</td>
<td>Graphics : pixel-based</td>
</tr>
<tr>
<td>MPEG, WMV, JPG, etc.</td>
<td>BMP, GIF, etc.</td>
</tr>
<tr>
<td>Natural attributes</td>
<td>Calculated attributes</td>
</tr>
<tr>
<td>*Digitized analog image</td>
<td>*Synthesized digital image</td>
</tr>
</tbody>
</table>

- With modern technology one can approach video performance and resolution, but the integrated effect of video is yet to be simulated in graphics.
- With increased HD video resolution, hardware acceleration is required; higher resolution – as well as demand for smooth 3D rendering – drives need for graphics acceleration.
Simple Graphics/Video Example

Video: Remotely captured/stored; locally decoded

Graphics: Locally generated

OMAP35x: ARM+DSP+Graphics

<table>
<thead>
<tr>
<th>OMAP35x Processor</th>
<th>OMAP3515</th>
<th>OMAP3530</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex A8</td>
<td>ARM® Cortex™ A8</td>
<td>C64x+ DSP &amp; video accelerator</td>
</tr>
<tr>
<td>C64x+ DSP</td>
<td>N/A</td>
<td>C64x+ DSP &amp; video accelerator</td>
</tr>
<tr>
<td>Video Accelerators (3525/3530)</td>
<td>2D/3D graphics accelerator – IMG SGX530</td>
<td>2D/3D graphics accelerator – IMG SGX530</td>
</tr>
<tr>
<td>Graphics (3515/3530)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display Subsystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera I/F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image Pipe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM C/SD/SDIO x3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Host Controller x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB 2.0 HS OTG Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GP MMCSDRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART w/IRDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mc BSP x5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>McSPI x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timers GP x12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDT x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image Pipe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel I/F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera I/F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphics (3515/3530)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDQ /1-wire</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 bit DAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Encoder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCD Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Accelerators (3525/3530)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D HW Accelerated = Fast UI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D in SW only = Slower UI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One of the highest performance graphics core in the market providing faster graphics performance without the need for an extra graphics card
- Off-loads tasks from ARM Cortex-A8 core and reduces system complexity

Also available on:
- AM3515
- AM3517
- DM3715
- DM3730
**What Processing Do You Need?**

**Accelerator: Audio/Video Algorithms**

**TMS320DM365 digital media processor**

**Performance**
- 1080p: 10fps - H.264
- 720p: 30fps
  - H.264 HP
  - MPEG4 SP
  - MPEG2 MP@ML
  - WMV9/VC-1 AP
- JPEG enc/dec at 50 MPix/sec

**Features**
- Core
  - ARM926EJ-S™ core, 300 MHz
  - H.264 coprocessor (HCP)
  - MPEG-4/JPEG coprocessor (MJCP)
  - Video Processing Subsystem (VPSS)
- Half power consumption in key modes
- Peripheral Highlights
  - EMAC 10/100
  - USB 2.0 HS OTG/mini-host w/PHY
  - External Memory Interface (EMIF)
  - Mobile DDR/DDR2
  - Keyscan/ADC
  - Audio Voice Codec

**Applications include:**
- IP security camera, multi-channel DVR, baby monitor, video doorbell, digital signage, personal media player, HD web cam

**TMS320DM6446: ARM9 + DSP + VICP**

**Cores**
- TMS320C64x+™ DSP Core at 810 MHz
- VICP Accelerator at 405 MHz
  - Real-Time Encode /Decode
  - Signal Processing APIs

**Performance**
- At 810MHz:
  - H.264 BP 720p30 Decode
  - Simultaneous H.264 BP D1 Enc/Dec
  - Dual H.264 BP D1 Decode
- At 594MHz:
  - H.264 BP D1 Encode
  - Simultaneous H.264 BP CIF Encode
  - H.264 MP 30fps D1 Decode
  - VCL/WMV9 D1 Decode
  - MPEG2 MP D1 Decode
  - MPEG4 ASP D1 Decode
  - Etc.

**Signal Processing Libraries...**
VICP Enables Additional DSP Algo Acceleration

◆ DM644x VICP Supports – examples:
  • Matrix Operations/Array Operations
    ◆ Matrix Multiplication/Transpose, Block Add/Average/Variance
    ◆ Array Multiplication/Addition/Fillmem, Array scalar operations
    ◆ Look Up Table
  • Digital Signal Processing Operations
    ◆ 1D, 2D FIR filtering
    ◆ Convolution, Correlation, FFTs
  • Digital Image and Video Processing Functions
    ◆ Alpha Blending, Color space Conversion
    ◆ Image rotation, Image Pack/Unpack
    ◆ Median Filtering

◆ Free library and user’s guide from TI:
  http://focus.ti.com/docs/toolsw/folders/print/sprc831.html

VICP: Video/Imaging Co-Processor

iMX (Imaging Extension)
◆ Color Space Conversion
◆ Filtering
◆ Interpolation
◆ Motion Estimation

VLCD* (Variable Length Codec Decoder)
◆ Huffman Coding / Decoding
◆ Quantization / Inverse Quant

Sequencer
◆ Control / Sequencing Tasks
◆ Buffer Management and Loop Control

Catalog Support
◆ VICP Utilization Included in TI and ASP Video and Imaging Codecs
◆ Accelerated library based on VICP
◆ Contact Local TI Sales Representative for further information
What Processing Do You Need?

VICP enable additional DSP headroom for customer application

DSP@900MHz Only, 720p 30fps

<table>
<thead>
<tr>
<th>Task</th>
<th>DSP Loading</th>
<th>DSP Headroom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Unpack</td>
<td>12.4%</td>
<td></td>
</tr>
<tr>
<td>Color Space Conversion</td>
<td>9.3%</td>
<td></td>
</tr>
<tr>
<td>Median Filter (5 Tap)</td>
<td>4.6%</td>
<td></td>
</tr>
<tr>
<td>Alpha Blending</td>
<td>3.7%</td>
<td></td>
</tr>
<tr>
<td>Image pack</td>
<td>12.4%</td>
<td></td>
</tr>
</tbody>
</table>

DSP Loading: 42.51%
DSP Headroom: 57.48%

DSP@900MHz + VICP@450MHz

<table>
<thead>
<tr>
<th>Task</th>
<th>DSP Loading</th>
<th>DSP Headroom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Unpack</td>
<td>VICP 17.9%</td>
<td></td>
</tr>
<tr>
<td>Color Space Conversion</td>
<td>VICP 31.2%</td>
<td></td>
</tr>
<tr>
<td>Median Filter (5 Tap)</td>
<td>VICP 7.9%</td>
<td></td>
</tr>
<tr>
<td>Alpha Blending</td>
<td>VICP 10.7%</td>
<td></td>
</tr>
<tr>
<td>Image pack</td>
<td>VICP 17.9%</td>
<td></td>
</tr>
</tbody>
</table>

DSP Loading: 0%
DSP Headroom: 100%

DM6446 Video Imaging Co-Processor (VICP)

- C64x+ DSP Core
- VICP

DM6446 Video Imaging Co-Processor (VICP)

- IMX
  - Parallel 8-MAC engine
  - Multi-level accumulators
  - Zero overhead looping control
  - Separate memory for Instruction/data
  - Shared Image Buffer

- VLC / VLD engine
  - JPEG, MPEG-1, MPEG-2, MPEG-4, H.263
  - Q/IQ engine
  - RVLC for error resiliency
  - H.264 CAVLD

- Simple 16-bit microcontroller
  - Can manage coprocessor interrupts / sync events
  - Can fully control algorithm processing
  - Data I/O, coprocessor and DSP execution
What Processing Do You Need?

**Applications**
- Transcoding (HD-HD, HD-SD), HD-video conf, networked DTV, HD-IP set-top boxes, video surveillance, video phones & digital media adaptors

**Core**
- ARM926EJ-S™ (MPU) at 300 MHz
- TMS320C64x+™ DSP Core at 600 MHz

**HD Coprocessors**
- Real-Time HD-HD Transcoding
  - Multi-format (mf) HD to mf HD / mf SD
  - Up to 2x real-time for HD-to-SD transcode
  - Real-time HD->HD transcoding for PVR

- Video Encode and Decode
  - HD 720p, H.264 BP encode
  - HD 1080i/p, H.264 MP@L4; VC1/WMV9; MPEG-2 MP@ML; MPEG-4 ASP; DivX dec
  - Simultaneous SD H.264 BP 30 fps enc/dec

- Video HW Assist
  - Resizer, 4:2:2<>4:2:0, Blending

**Peripheral Highlights**
- Video ports
  - 2x8-bit BT.656 or 1x16-bit BT.1120 capture
  - 2x8-bit BT.656 or 1x16-bit BT.1120 display

**HD Davinci™ - DM6467**
HD Video Transcode, Enc/Dec App Processing

**ARM Subsystem**
- ARM926EJ-S CPU 300 MHz

**DSP Subsystem**
- C64x+™ DSP Core 600 MHz

**Switched Central Resource (SCR)**

**DM6467 HDVICP**

Two Video accelerators on DM646X
- - HD-VICP0 - Video Decoding Accelerator
- - HD-VICP1 - Video Encoding Accelerator

**Sequencer**
- Responsible for firing the co-processors and the DMA commands.
- Prepares Commands for co-processors

**Host processor (C64x+)**
- Responsible for initializations at frame and slice levels
- Prepares the data for the execution of MB’s for the current chunk and messages the ARM968.
- Available for other compute intensive tasks.
What Processing Do You Need?

OMAP35x Processors: ARM+DSP+Graphics

OMAP3525
- ARM® Cortex™ A8
- C64x+ DSP & video accelerator
- LPDDR@166MHz
- L2 256KB
- L1P 16KB
- L1D 16KB
- LPDDR@166MHz
- Neon float support
- MPEG4 720p 24fps/30fps encode/decode
- H.264 MP VGA decode
- H.264 BP/VC1/WMV9 D1 encode/decode
- Neon float support

OMAP3530
- ARM® Cortex™ A8
- C64x+ DSP & video accelerator
- LPDDR@166MHz
- L2 256KB
- L1P 16KB
- L1D 16KB
- LPDDR@166MHz
- Neon float support
- MPEG4 720p 24fps/30fps encode/decode
- H.264 MP VGA decode
- H.264 BP/VC1/WMV9 D1 encode/decode
- Neon float support

OMAP35XX - IVA2.2

iME: Motion Estimation
- H264, WMV9, Real9, MPEG-4
- Specialized instruction for Motion Estimation
- 16 Pixel SADs per cycle, no data alignment constraints
- Programmable machine but no loop and no condition

iVLCD: Variable Length Coding/Decoding
- H264 (BP only), WMV9, MPEG-4
- CAVLC
- Works with internal memory
- H264 CAVLC 4Mbps:

iLF: MB Loop Filtering
- H264, WMV9, Real9, MPEG-4
- One 8-tap filter per cycle
- Programmable machine, no loop and no condition
- Handle BS and Pixel-deps
- H264 LF Luma+Chroma
Accelerators: Video Port (VPSS) / Display (DSS)

The Video Port SubSystem (from DaVinci team) and Display SubSystem (from OMAP3) each contain a number of hardware accelerators to off-load a variety of video-related tasks from the ARM and/or DSP CPU’s.

VPSS (Video Port SubSystem)

DaVinci Video Port Accelerators (VPSS)
**What Processing Do You Need?**

### Front End: Resizer, Previewer, H3A

- **CCDC**
- **Previewer**
- **Resizer**

**H3A**
- Statistical engine for calculating image properties
- Histogram:
  - Histogram data collection (in RGB color space)
  - ARM + DSP can access these statistics
- Automatic Focus Control
- Automatic White Balance Correction
- Automatic Exposure Compensation

**Previewer**
- Bayer RGB to YCbCr 4:2:2 color space conversion
- Programmable noise filter
- Offloads processing effort

**Resizer**
- 4x to 1/4x Resizing N/256 Zoom step
- Linear and Bi-Cubic Resize Algorithms
- Automatic Video Rescale
- Offloads processing effort

### Back End: On-Screen Display (OSD)

- **OSD**
- **VENC**

**Hardware On-Screen Display (OSD)**
- 2 separate video windows
- 2 separate OSD windows
  - One can be used as attribute window for alpha-blending between video and OSD windows
- 1 rectangular cursor window
- 1 background color
What Processing Do You Need?

**OSD Usage: Set-Top Box Example**

- **Video0** - Background
- **Video1** – Overlay (e.g. PIP)
- **OSD0** – on-screen menu
- **OSD1** – alpha-blending/pixel-by-pixel OSD attribute
- **Cursor** – as selection

**OSD Attribute Window**

- Allows Pixel by Pixel Blending of OSD0 and Video Windows
- Uses a 4-bit, Bit-map Window

Blending

8-level blending

- 000: 00.0%, 100% Video
- 001: 12.5%, 87.5% Video
- 010: 25.0%, 75.0% Video
- ...
- 110: 75.0%, 25.0% Video
- 111: 100%, 00.0% Video
**DSS (Display SubSystem)**

**OMAP Display Sub-System : High-Level Diagram**

- **DDR2 / SDRAM**
- **L3 System Interconnect**
- **Display DMA**
  - Graphics (FIFO)
  - Video 1 (FIFO)
  - Video 2 (FIFO)
- **Graphics Pipeline**
  - Gamma correction / Color Palette Operation
- **Video 1 Pipeline**
  - Color Space Conversion, Up/Down Scaling
- **Video 2 Pipeline**
  - Color Space Conversion, Up/Down Scaling
- **Overlay Mgr's**
- **Overlay Output Path**
- **LCD Output Path**
- **TV Output Path**

**OMAP (DSS) Overlay Example**

- Background
- Video 1
- Video 2
- Graphics

![Image of OMAP (DSS) Overlay Example](https://example.com/image.png)
OMAP (DSS) Overlay Manager
Color Key Example

Top Layer
(Video 1 or 2)

Bottom Layer
(Graphics)

Source
Transparency
Color Key

Screen

OMAP Display Overlay Manager

Overlay Optimization
- Only fetch needed pixels from memory
  - At least video window 1 and graphics window must be enabled
  - The graphics pixels under the video 1 will not be fetched from the memory
  - The transparency color key must be disabled
- Reduces the peak bandwidth
  - Only visible pixels from graphics and video buffers are fetched and displayed

Overlay Manager

Memory

Graphics

Video 1

Video Conferencing
Hardware Accelerator Summary

<table>
<thead>
<tr>
<th>Devices</th>
<th>3D Graphics</th>
<th>Video/Imaging/Audio</th>
<th>Video Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM355</td>
<td></td>
<td>MJCP, JPEG, MPEG4</td>
<td>OSD, Resizer, Previewer, H3A</td>
</tr>
<tr>
<td>DM365/368</td>
<td></td>
<td>MJCP &amp; HCP: JPEG, MPEG4, H264, VC1, Facial; MP3, AAC, WMA, AEC</td>
<td>OSD, Resizer, Previewer, H3A</td>
</tr>
<tr>
<td>AM3517</td>
<td>PowerVR SGX</td>
<td></td>
<td>OSD, Resizer, Color Key</td>
</tr>
<tr>
<td>AM3715</td>
<td>PowerVR SGX</td>
<td></td>
<td>OSD, Resizer, Color Key</td>
</tr>
<tr>
<td>DM6446</td>
<td></td>
<td>VICP: Video, Image, Other</td>
<td>OSD, Resizer, Previewer, H3A</td>
</tr>
<tr>
<td>DM6467</td>
<td></td>
<td>HDVICP: Video Encoding</td>
<td>TSIF</td>
</tr>
<tr>
<td>OMAP3515</td>
<td>PowerVR SGX</td>
<td>IVA2.2: Video Encoding</td>
<td>OSD, Resizer, Overlay Mgr</td>
</tr>
<tr>
<td>OMAP3525</td>
<td>PowerVR SGX</td>
<td>IVA2.2: Video Encoding</td>
<td>OSD, Resizer, Overlay Mgr</td>
</tr>
<tr>
<td>OMAP3530</td>
<td>PowerVR SGX</td>
<td>IVA2.2: Video Encoding</td>
<td>OSD, Resizer, Overlay Mgr</td>
</tr>
<tr>
<td>DM3725</td>
<td></td>
<td>IVA2.2: Video Encoding</td>
<td>OSD, Resizer, Overlay Mgr</td>
</tr>
<tr>
<td>DM3730</td>
<td>PowerVR SGX</td>
<td>IVA2.2: Video Encoding</td>
<td>OSD, Resizer, Overlay Mgr</td>
</tr>
</tbody>
</table>

Note: “Hardware accelerated” algorithms require software support – which is provided by TI and TI 3Ps
Between the enormity of examining the features of every one of these peripherals, and the fact that each device has a different subset of peripherals, we just don’t have the time to dig into each one of them.

For this reason, we’ll only take a brief look at three of them:

- PRU – Programmable Real-time Unit
- SCR – Switched Central Resource (i.e. internal bus crossbar)
- EDMA3 – Enhanced DMA controller (version 3)

The first two are probably not very self-explanatory. The third, conveniently, is part of the SCR discussion.
**PRU – Programmable Real-time Unit**

**Programmable Realtime Unit (PRU)**

**PRU consists of:**
- 2 Independent, Realtime RISC Cores
- Access to pins (GPIO)
- Its own interrupt controller
- Access to memory (master via SCR)
- Device power mgmt control (ARM/DSP clock gating)

**Use as a soft peripheral to implement additional on-chip peripherals**

**Examples implementations include:**
- Soft UART
- Soft CAN

**Create custom peripherals or setup non-linear DMA moves.**

**Implement smart power controller:**
- Allows switching off both ARM and DSP clocks
- Maximize power down time by evaluating system events before waking up DSP and/or ARM

**PRU SubSystem : IS / IS-NOT**

<table>
<thead>
<tr>
<th>Is</th>
<th>Is-Not</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dual 32-bit RISC processor</strong> specifically designed for manipulation of packed memory mapped data structures and implementing system features that have tight real time constraints.</td>
<td>Is not a H/W accelerator used to speed up algorithm computations.</td>
</tr>
</tbody>
</table>
| **Simple RISC ISA:**
  - Approximately 40 instructions
  - Logical, arithmetic, and flow control ops all complete in a single cycle | Is not a general purpose RISC processor:
  - No multiply hardware/instructions
  - No cache or pipeline
  - No C programming |
| **Simple tooling:**
  - Basic command-line assembler/linker | Is not integrated with CCS. Doesn't include advanced debug options |
| **Includes example code** to demonstrate various features. Examples can be used as building blocks. | No Operating System or high-level application software stack |
Moving Data Around – SCR / EDMA3

DM644x Architecture

Masters
- ARM
- C64+ DSP
- VPSS
- EDMA3
- USB
- EMAC
- ATA
- VLYNQ

Slaves
- ARM Memory
- DSP Memory
- DDR2
- Async EMIF
- MMC / SD
- ASP
- UART x3
- PWM x3
- SPI
- IIC
- Timers x3

Crossbar Connections:
- Bridge 1
- Bridge 2
- Bridge 3
- Bridge 4
- Bridge 5

Connections:
- IAHB
- DAHB
- PB
- DB
- VPFE
- VPBE
- TC0
- TC1
- SCR 1
- SCR 5
- SCR 3 – CFG xbar

Components:
- DDR
- CFG
- CFG 0
- CFG 1
## Final Considerations

### Memory Map & MMU

#### DM6446 Memory Map

<table>
<thead>
<tr>
<th>ARM</th>
<th>C64x+ DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K</td>
<td>ARM's Internal RAM</td>
</tr>
<tr>
<td>176K</td>
<td>Reserved</td>
</tr>
<tr>
<td>4M</td>
<td>DSP CFG Space</td>
</tr>
<tr>
<td>4M</td>
<td>CFG Bus Peripherals</td>
</tr>
<tr>
<td>128M</td>
<td>EMIFA Prog/Data</td>
</tr>
<tr>
<td>16K</td>
<td>Reserved</td>
</tr>
<tr>
<td>176K</td>
<td>DSP's Internal Memory</td>
</tr>
<tr>
<td>32K</td>
<td>DDR2 Control</td>
</tr>
<tr>
<td>128M</td>
<td>Reserved</td>
</tr>
<tr>
<td>256M</td>
<td>DDR2</td>
</tr>
</tbody>
</table>

**Shared Memory**
- DDR2
- EMIFA
- DSP L2RAM Shadow
- DSP L1P/D Shadow
- ARM RAM0/1 Shadow

**Notes**
- CPU's share buffers by simply passing pointers
- Memory can be allocated by users as best fits ARM/DSP applications

Looking at how we've used DDR2...

#### DVEVM Default DDR2 Memory Map

- **Linux**
  - 0: Linux
  - 120M: CMEM
  - 8M: DDR heap (DSP heap)

- **DSP linker segments**
  - 122M: CMEM
  - 4M: DDR
  - 1M: DSPLINKMEM
  - 128B: CTRLRESET

When booted with MEM=120M

CMEM = Continuous Memory

Shared buffers: ARM ↔ DSP

(Size set by loadmodules.sh)

Exclusively used for the DSP’s dynamically allocated memory

Code, Stack, and Static Data (.tcf)

(loadmodules.sh = ddr_start/ddr_size)

Memory for DSPLINK

Memory for Reset Vectors
**MMU**

**DM6446 Memory Mgmt Unit (MMU)**

- ARM's memory accessed thru MMU
  - Benefits: virtual memory, memory protection
  - MMU discussed briefly in Chapters 8 and 11
- DSP has no MMU (only memory protection which is not commonly used)
  - DSP can access ARM's memory – easy buffer sharing (but less robust?)
  - No MMU in memory path increases performance
  - Address translation req'd during buffer passing from ARM to DSP – thankfully, Codec Engine performs this task

**OMAP35x MMU's**

- ARM's memory still accessed thru MMU
- DSP now has an MMU in its memory path
  - Common usage: “Set & Forget”
  - Setting it only once at DSP loadtime protects Linux-space while minimizing MMU performance degradation
  - Again, Codec Engine framework sets the MMU for us at loadtime
## Access to Peripherals

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>ARM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPSS (Video capture/display)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>EMAC/MDIO (Ethernet)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>USB 2.0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ARM Interrupt Controller</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ATA/CF, MMC/SD</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPIO, PWM0/1/2, Watchdog Timer</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>I2C, SPI, UART0/1/2</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>EDMA</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Timer0/1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ASP (Audio Serial Port)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DDR2, EMIF</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Some devices provide full access to all peripherals from the ARM and DSP
- ARM-only access: OMAP35, DM37x, DM6467, DM644x
- ARM/DSP access: OMAP-L137, OMAP-L138
Pin Muxing

**What is Pin Multiplexing?**

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

**Pin Muxing Tools**

- Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- Provides Pin Mux Register Configurations
To force proper pagination, this page was intentionally left almost blank.
Device Families
(Sitara, C6000, DaVinci)

Processor Families Evolution

ARM and ARM+DSP/Video

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>General Purpose</th>
<th>Video Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Only</td>
<td>ARM+DSP</td>
<td>ARM / ARM+DSP</td>
</tr>
<tr>
<td>ARM926</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM1705</td>
<td>OMAP-L137</td>
<td>DM355</td>
</tr>
<tr>
<td>AM1707</td>
<td></td>
<td>DM365</td>
</tr>
<tr>
<td>AM1806</td>
<td>OMAP-L138</td>
<td>DM644x</td>
</tr>
<tr>
<td>AM1808</td>
<td>(also C6L13x)</td>
<td>DM6467</td>
</tr>
<tr>
<td>Cortex A8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMAP3503</td>
<td></td>
<td>OMAP3525</td>
</tr>
<tr>
<td>OMAP3515</td>
<td></td>
<td>OMAP3530</td>
</tr>
<tr>
<td>AM3505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM3515</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM3703</td>
<td></td>
<td>DM3725</td>
</tr>
<tr>
<td>AM3715</td>
<td></td>
<td>DM3730</td>
</tr>
</tbody>
</table>

Pin-for-Pin Compatibility

Sitara (AM)  ""  DaVinci (DM)
Choosing a Device: Web Tool

DSP & ARM MPU Selection Tool

Introduction

At this point we have seen an overview of the DaVinci and OMAP devices. Next we will explore the basic components of the DaVinci / OMAP software model. Each of the concepts outlined here will be discussed in further detail in succeeding chapters, but in this chapter we hope to give you a look at the big picture.

Learning Objectives

<table>
<thead>
<tr>
<th>Linux Applications</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Driver Interface (Linux)</td>
</tr>
<tr>
<td></td>
<td>VISA Interface (Codec Engine)</td>
</tr>
<tr>
<td>Algorithms/Codecs (Signal Processing)</td>
<td>xDAIS (eXpressDSP™ Algorithm Standard)</td>
</tr>
<tr>
<td></td>
<td>xDM (extending xDAIS with Classes)</td>
</tr>
<tr>
<td></td>
<td>VISA vs xDM</td>
</tr>
<tr>
<td>Codec Engine Details</td>
<td>Where to Get Your Algorithms</td>
</tr>
<tr>
<td>Software Summary</td>
<td>ARM or ARM+DSP</td>
</tr>
<tr>
<td></td>
<td>RPC - Remote Procedure Call</td>
</tr>
<tr>
<td></td>
<td>Code Review</td>
</tr>
</tbody>
</table>
# Chapter Topics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction to DaVinci / OMAP Software</strong></td>
<td>2-1</td>
</tr>
<tr>
<td>Application Layer</td>
<td>2-3</td>
</tr>
<tr>
<td>Concepts</td>
<td>2-3</td>
</tr>
<tr>
<td>I/O Layer - Linux Device Drivers</td>
<td>2-6</td>
</tr>
<tr>
<td>Access Signal Processing Horsepower using the Codec Engine’s VISA Interface</td>
<td>2-7</td>
</tr>
<tr>
<td>Adding VISA to our “Master Thread”</td>
<td>2-9</td>
</tr>
<tr>
<td><strong>Signal Processing Layer – Algorithms and Codecs</strong></td>
<td>2-10</td>
</tr>
<tr>
<td>What is a “Codec”?</td>
<td>2-10</td>
</tr>
<tr>
<td>xDAIS (eXpressDSP™ Algorithm Standard)</td>
<td>2-11</td>
</tr>
<tr>
<td>xDM (Extending xDAIS with Algorithm Classes)</td>
<td>2-13</td>
</tr>
<tr>
<td>VISA vs. xDM</td>
<td>2-14</td>
</tr>
<tr>
<td><strong>Where to Get Your Algorithms</strong></td>
<td>2-17</td>
</tr>
<tr>
<td><strong>CODEC Engine Details</strong></td>
<td>2-19</td>
</tr>
<tr>
<td>RPC – Remote Procedure Call</td>
<td>2-20</td>
</tr>
<tr>
<td>Code Review</td>
<td>2-22</td>
</tr>
<tr>
<td><strong>Software Summary</strong></td>
<td>2-26</td>
</tr>
<tr>
<td><strong>For More Information</strong></td>
<td>2-28</td>
</tr>
</tbody>
</table>
Application Layer

Concepts

Goal: Accelerated Time to Market

New Product Idea

Create SW foundation → Product differentiation

The DaVinci™ Effect

Leverage DaVinci SW foundation → Product differentiation

Shorter development cycle and/or

Leverage DaVinci SW foundation → More time for product differentiation

Digital Multimedia System: Needs

GUI

User interface

Input (Driver)

Master Thread

user code

Process (algorithm)

Output (Driver)

GP OS (eg: Linux)

- Manage multiple threads
- Assist in creation of user interface
- Manage memory
- Provide system services

import source data

Convert input data to desired results

export results
**Linux-based Solution**

**User Space**

```
// "Master Thread"
// Create Phase
get IO
alloc process RAM
// Execute Phase
while(run)
  Input (exch bufs)
  Process
  Output (exch bufs)
// Delete Phase
free process RAM
surrender IO
```

**Kernel Space**

**GUI**
- Song
- Volume
- Bass
- Treble

**Process (algorithm)**

---

**DaVinci Framework API: VISA & Drivers**

**User Space**

```
// "Master Thread"
// Create Phase
get IO
alloc process RAM
// Execute Phase
while(run)
  Input (exch bufs)
  Process
  Output (exch bufs)
// Delete Phase
free process RAM
surrender IO
```

**Kernel Space**

**GUI**
- Song
- Volume
- Bass
- Treble

**Process (algorithm)**

---

**VISA API**
- create
- process
- control
- delete

**Driver API**
- open
- read
- write
- ioctl
- close

**VISA**
- Video, Imaging,
- Speech, Audio
  (same for every OS)

**Driver**
- Varies from
  OS-to-OS
DaVinci technology software development is divided into three areas:

1. Application programs
2. Device I/O (drivers)
3. Algorithms (Signal processing)

Goal: abstract details of signal processing algorithms and drivers so application programmer can quickly author a differentiated system
I/O Layer - Linux Device Drivers

Linux Device Drivers

- **Driver API’s**
  - Collection of drivers chosen to support the I/O ports for each O/S
  - Where available, common O/S drivers are chosen (Linux - see below)
  - Where no common driver exists, TI must create its own driver API

- **Linux Drivers**
  - **Storage** - ATA/IDE, NAND, NOR, MMC/SD
  - **Audio** - ALSA, OSS Audio driver
  - **Video** - V4L2 for Capture/Display, FBDev for OSD graphics, IPIPE, Resizer, Previewer, H3A
  - **Network** - Ethernet
  - **USB** - Mass storage - Host and Gadget drivers
  - **Serial** - UART, I2C, SPI
  - **Other** - PWM, Watchdog, EDMA, GPIO
  - **Boot** - Das U-Boot (open source Linux boot-loader)

Linux – Basic File I/O & Character Driver API

Basic Linux file I/O usage in user programs is via these API:

```c
myFileFd = fopen("/mnt/harddrive/myfile", "rw");
fread ( aMyBuf, sizeof(int), len, myFileFd );
fwrite( aMyBuf, sizeof(int), len, myFileFd );
fclose( myFileFd );
```

Additionally, you can use `fprintf()` and `fscanf()` for more feature-rich file read/writes.

Simple drivers use the same format as files...

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

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

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

- Basic drivers will be covered in more detail in Chapter 6.
- Some Linux drivers (such as V4L2 and FBDEV video drivers) typically use `mmap` and `ioctl` commands instead of `read` and `write` that pass data by reference instead of by copy. These will be studied in greater detail in the Chapter 7.
**Master Thread – Accessing I/O**

```c
idevfd = open("/dev/xxx", O_RDONLY);
ofilefd = open("./fname", O_WRONLY);
ioctl(idevfd, CMD, &args);

while( doRecordVideo == 1 ) {
    read(idevfd, &rd, sizeof(rd));
    write(ofilefd, &wd, sizeof(wd));
}
close(idevfd);
close(ofilefd);
```

// **Create Phase**
// get input device
// get output device
// initialize IO devices...

// **Execute phase**
// read/swaps buffer with Input device

// **Delete phase**
// return IO devices back to OS

---

**Access Signal Processing Horsepower using the Codec Engine’s VISA Interface**

**VISA – Four SPL Functions**

- **Create**: creates an instance of an algo that is, it malloc's the required memory and initializes the algorithm
- **Process**: invokes the algorithm
  calls the algorithms processing function passing descriptors for in and out buffers
- **Control**: used to change algo settings
  algorithm developers can provide user controllable parameters
- **Delete**: deletes an instance of an algo
  opposite of “create”, this deletes the memory set aside for a specific instance of an algorithm

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  _create _delete _process _control

- Linux/ARM Programmer
  `create()` `control()` `process()` `delete()`
VISA – Eleven Classes

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  - _create
  - _delete
  - _process
  - _control

- VISA = 4 processing domains:
  - Video Imaging Speech Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  - VISA Encoders/Decoders
  - Video ANALYTICS & TRANSCODE
  - Universal (generic algorithm if!) New!

- VISA API

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  - _create
  - _delete
  - _process
  - _control

- VISA = 4 processing domains:
  - Video Imaging Speech Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  - VISA Encoders/Decoders
  - Video ANALYTICS & TRANSCODE
  - Universal (generic algorithm if!) New!

- TI’s CODEC engine (CE) provides abstraction between VISA and algorithms
- Application programmers can purchase xDM algorithms from TI third party vendors
  - or, hire them to create complete SPL soln’s
- Alternatively, experienced DSP programmers can create xDM compliant algos (discussed next)
- Author your own algos or purchase depending on your DSP needs and skills

Complexities of Signal Processing Layer (SPL) are abstracted into four functions:

- Video Imaging Speech Audio

Separate API set for encode and decode thus, a total of 11 API classes:

- Video ENC/DEC
- Universal

VISA = 4 processing domains:

- Video Imaging Speech Audio

Separate API set for encode and decode thus, a total of 11 API classes:

- Video ANALYTICS & TRANSCODE
- Universal (generic algorithm if!) New!

Reducing dozens of functions to 4
VISA Benefits

**Application Author Benefits**
- App author enjoys benefits of signal processing layer without need to comprehend the complexities of the DSP algo or underlying hardware
- Application author uses only one API for a given media engine class
- Changing CODEC within the class involves no changes to app level code
- All media engine classes have a similar look and feel
- Adapting any app code to other engines and API is very straightforward
- Example apps that use VISA to manage xDM CODECs provided by TI
- Customers can create multimedia frameworks that will leverage VISA API
- VISA contains hooks allowing additional functionalities within CODECs
- Authoring app code, multimedia frameworks & end equipment expertise is what customers do best, and want to focus on - VISA optimizes this

**Algorithm Author Benefits**
- CODEC engine authors have a known standard to write to
- CODEC authors need no knowledge of the end application
- CODECs can be sold more readily, since they are easy to apply widely
- Each class contains the information necessary for that type of media
- VISA, and xDAIS-DM, build on xDAIS – an established algo interface
- Tools exist today to adapt algos to xDAIS, and may include –DM soon (?)

Adding VISA to our “Master Thread”:

**Master Thread Key Activities**

```c
idevfd = open("/dev/xxx", O_RDONLY);  // Create Phase
ofilefd = open("./fname", O_WRONLY);  // get input device
ioctl(idevfd, CMD, &args);            // get output device
myCE = Engine_open("vcr", myCEAttrs); // initialize IO devices...
myVE = VIDENC_create(myCE, "videnc", params);  // prepare VISA environment

while( doRecordVideo == 1 ) {  // prepare to use video encoder
    read(idevfd, &rd, sizeof(rd));
    VIDENC_process(myVE, ...);
    // VIDENC_control(myVE, ...);
    write(ofilefd, &wd, sizeof(wd));
}

close(idevfd);  // return IO devices back to OS
close(ofilefd);
VIDENC_delete(myVE);  // algo RAM back to heap
Engine_close(myCE);  // close VISA framework
```

- See Chapter 9 for more details of VISA functions (i.e. prototypes)
Signal Processing Layer – Algorithms and Codecs

What is a “Codec”?

Signal Processing Layer : What is a Codec?

1. Compression / Decompression algorithm
2. Single device containing both an analog-to-digital (A/D) and digital-to-analog (D/A) converter
3. In the world of TI's DaVinci software, we often use the term “codec” to refer to any real-time algorithm

Should the framework have been called the “Algo Engine”, or how about the “xDAIS Engine”?

In the end, “Codec Engine” won out.
xDAIS (eXpressDSP™ Algorithm Standard)

“Plugging-in” xDAIS Algorithms

- For ease of use – and to enable automation – algorithms need a to conform to a standardized interface
- xDAIS provides a time-tested, real-time protocol (used by the Codec Engine)
- xDAIS algos are similar to C++ classes in that they don't occupy memory until an instance is created; therefore, they provide three interfaces:
  - Create (i.e. constructor) methods
  - Process method(s)
  - Delete methods
- Unlike C++, though, algorithms don't allocate their own memory; rather, resource mgmt is reserved for the System Integrator (via Codec Engine config)

xDAIS/xDM for Algorithm Authors

- xDAIS provides a consistent, straight-forward set of methods for algorithm authors to specify an algorithm’s resource needs ... 
  ... therefore, using xDAIS compliant algos allows users to easily manage resources, such as system memory, DMA, and accelerators via a consistent API
- Algorithms cannot take resources, but must request them – this is done thru standard xDAIS required functions
- Codec Engine's VISA classes match up to xDM classes – xDM (xDAIS for Digital Media) is an extension to xDAIS which defines the plug-n-play multimedia algorithm classes
- Similar to C++ classes, algo's shouldn't use global variables, but rather bundle them into an instance object (i.e. class object)

Required Algorithm Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance Creation</strong></td>
<td></td>
</tr>
<tr>
<td>(i.e. Constructor)</td>
<td></td>
</tr>
<tr>
<td>e.g. VIDENC_create()</td>
<td></td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td></td>
</tr>
<tr>
<td>VIDENC_process()</td>
<td></td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td></td>
</tr>
<tr>
<td>VIDENC_delete()</td>
<td></td>
</tr>
<tr>
<td>Specify Memory requirements</td>
<td>(algAlloc)</td>
</tr>
<tr>
<td>Specify DMA requirements</td>
<td>(dmaAlloc)</td>
</tr>
<tr>
<td>Initialize algorithm</td>
<td>(algInit, dmalInit)</td>
</tr>
<tr>
<td>Prepare scratch memory</td>
<td>(algActivate)</td>
</tr>
<tr>
<td>Run algorithm</td>
<td>(process)</td>
</tr>
<tr>
<td>Save scratch data</td>
<td>(algDeactivate)</td>
</tr>
<tr>
<td>Free resources</td>
<td>(algFree, dmaFree)</td>
</tr>
</tbody>
</table>
iAlg Functions Summary

♦ **Create Functions** (i.e. Constructor Functions)
  - **algNumAlloc** - Tells application (i.e. CODEC engine) how many blocks of memory are required; it usually just returns a number
  - **algAlloc** - Describes properties of each required block of memory (size, alignment, location, scratch/persistent)
  - **algInit** - Algorithm is initialized with specified parameters and memory

♦ **Execute Functions**
  - **algActivate** - Prepare scratch memory for use; called prior to using algorithms process function (e.g. prep history for filter algo)
  - **algDeactivate** - Store scratch data to persistent memory subsequent to algo’s process function
  - **algMoved** - Used if application relocates an algorithm’s memory

♦ **Delete Function**
  - **algFree** - Algorithm returns descriptions of memory blocks it was given, so that the application can free them
**xDM (Extending xDAIS with Algorithm Classes)**

**xDM: Extending xDAIS**

- Digital Media extensions for xDAIS “xDAIS-DM” or “xDM”
- Enable plug + play ability for multimedia CODECs across implementations / vendors / systems
- **Uniform** across domains…video, imaging, audio, speech
- Can be extended for custom / vendor-specific functionality
- **Low overhead**
- Insulate application from component-level changes
  - Hardware changes should not impact software
  - PnP …enable ease of replacement for versions, vendors
- **xDM is still Framework and O/S Agnostic**
- Create code faster – Published API enables early **and** parallel development
  - System level development in parallel with component level algo development
  - Reduces integration time for system developers
- **Published and Stable interface helps:**
  - TI, third parties, and customers
  - Support backward compatibility

**Eleven xDM Classes**

- **VIDENC** (encode)
- **VIDDEC** (decode)
- **VIDANALYTICS** (analysis)
- **VIDTRANSCODE** (transcode)
- **IMGENC** (encode)
- **IMGDEC** (decode)
- **SPHENC** (encode)
- **SPHDEC** (decode)
- **AUDENC** (encode)
- **AUDDEC** (decode)

- Create your own VISA compliant algorithm by inheriting the Universal class
- Then, use your algorithm with the Codec Engine, just like any other xDM algo
VISA vs. xDM

VISA – Application Programmer

VISA API Layer: Application Programmer

VIDDEC_create()  VIDDEC_control()  VIDDEC_process()  VIDDEC_delete()

---

xDM – Algorithm Author

xDM API Layer: Algorithm Author

algNumAlloc  algAlloc  algInit  algFree

---------  ---------  --------  --------

---

algActivate  algDeactivate  process  control

---------  ---------  --------  -------

---
VISA – CODEC Engine – xDM

VISA API Layer: Application Programmer

- VIDDEC_create()
- VIDDEC_control()
- VIDDEC_process()
- VIDDEC_delete()

CODEC Engine framework: TI

- algNumAlloc
- algAlloc malloc()
- algInit
- control
- activate
- process deactivate
- algNumAlloc
- algFree free()

xDM API Layer: Algorithm Author

- algNumAlloc
- algAlloc
- algInit
- algFree

- algActivate
- algDeactivate
- process
- control

VISA – Codec Engine – xDM

VISA API Layer: Application Programmer

- VIDDEC_create()
- VIDDEC_control()
- VIDDEC_process()
- VIDDEC_delete()

Codec Engine is provided by TI
You need only be concerned with VISA or xDM

xDM API Layer: Algorithm Author

- algNumAlloc
- algAlloc
- algInit
- algFree

- algActivate
- algDeactivate
- process
- control
Almost blank ...
Where to Get Your Algorithms

1. **Make your own** xDM algorithms  
   (discussed in Chapter 13)

2. Obtain free algorithms/CODEC's from TI  
   (www.ti.com/dms)

3. Purchase **Individual CODECs**  
   from TI or TI Third Party

4. Contract a **Complete DSP Executable**  
   from one of TI’s ASP’s

**Algorithm / Codec Inventory**

<table>
<thead>
<tr>
<th>Codec/Ring</th>
<th>TMD320DM64x</th>
<th>TMD320DM64x</th>
<th>TMD320DM643x</th>
<th>TMD320DM640</th>
<th>TMD320DM63x</th>
<th>OMAP33x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video &amp; Imaging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H.264 Video Decoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H.264 Video Encoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCIIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG Imaging Decoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG Imaging Encoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-2 Video Decoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-2 Video Encoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-4 Video Decoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-4 Video Encoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCI1 Video Decoder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Digital Media Software: [www.ti.com/dms](http://www.ti.com/dms)
Available eXpressDSP™ Compliant Algorithms

- More than 1000 TMS320 algorithms provided by TI third parties
- eXpressDSP Compliance Program available for greater confidence
- Ensures:
  - Interoperability
  - Portability
  - Maintainability
  - Measurability
- eXpressDSP™ Algorithm Standard Developer’s Kit available to help develop your own compliant IP
CODEC Engine Details

DaVinci Technology Framework: ARM Only

- xDM algo on ARM:
  - Familiar layout to 1000’s of Linux programmers
  - Optimal for low to medium demand algos

- Will all algos run successfully on the ARM?
  - MIPS
  - Power efficiency
  - Separate I/O interruptions from DSP processing
  - Non-determinism of GP OS’s

- So, in many cases, hard real-time high demand DSP work needs a DSP to implement the processing phase of the system

- How much extra work will be imposed on the application author to locate the xDM algo on DaVinci technology-based DSPs?

DaVinci Technology Framework: ARM + DSP

- xDM algo on DSP:
  - ARM master thread app code unchanged
  - xDM algo same as one used on ARM
  - CODEC engine abstracts all detail from user
RPC – Remote Procedure Call

Local and Remote Algo Objects

How does a VISA_process call know whether to call a local function or to call a remote function via a stub?

VISA_object for local engine

(*create) → VISA_create
(*control) → xDAIS DM control
(*process) → xDAIS DM process
(*delete) → VISA_delete

VISA_object for remote engine

(*create) → VISA_create
(*control) → Stub function
(*process) → Stub function
(*delete) → VISA_delete

VISA_create and VISA_delete test a bit in the engine object to determine if engine is local or remote and proceed accordingly.

Local Call of Algo From Application

A local call is simply a function call – easy since all the data resides on the same CPU.
Remote Procedure Call “RPC”

- The CODEC engine abstracts remote calls
- Stub functions marshall (i.e. gather together) the required arguments
- Skeletons unpack args, call the algo on the remote processor

DaVinci Technology System
SW Component Details

Application Code
- No changes for local vs remote algo
- Serves as ‘master thread’
- Controls all other components in system

Algorithm / DSP Task
- No change to algo code to run on DSP
- No change to algo code
- DSP Task is a pure ‘data transducer’:
  - no direct control over peripherals
  - ‘slave’ to app code control
  - not the ‘master’ of the application
- Algo inside TSK to provide priority, context
- Algo can use ACPY, DMAN to bring buffer data from shared mem to local RAM

IO Drivers
- No change to peripherals
- No change to application code
- Drivers accessible to Linux community
- Data does not pass through app on way to algo – no extra layer of overhead
- Buffers are in shared memory, equally accessible to DSP

Interprocessor Communication
- CODEC engine abstracts all IPC details
- App/algo unaware of location of algo
- Infrastructure provided by TI for DaVinci technology
Code Review

CODEC Engine: CERuntime_init()

- `idevfd = open(...);` // create
- `ofilefd = open(...);`
- `ioctl(...);`
- `CERuntime_init();`
- `myCE = Engine_open(...);`
- `myVE = VIDENC_create(...);`
- `while( doSvc ){ // execute`
  - `read(idevfd, ...);`
  - `VIDENC_control(myVE, ...);`
  - `VIDENC_process(myVE, ...);`
  - `write(ofilefd, ...);`}
- `close(idevfd); // delete`
- `close(ofilefd);`
- `VIDENC_delete(myVE);`
- `Engine_close(myCE);`

**CERuntime_init()**: 
- Create-phase activity
- Creates the CODEC engine thread
- Only needs to be done once in a system

CODEC Engine: Engine_open()

- `idevfd = open(...);` // create
- `ofilefd = open(...);`
- `ioctl(...);`
- `CERuntime_init();`
- `myCE = Engine_open(...);`
- `myVE = VIDENC_create(...);`
- `while( doSvc ){ // execute`
  - `read(idevfd, ...);`
  - `VIDENC_control(myVE, ...);`
  - `VIDENC_process(myVE, ...);`
  - `write(ofilefd, ...);`}
- `close(idevfd); // delete`
- `close(ofilefd);`
- `VIDENC_delete(myVE);`
- `Engine_close(myCE);`

**Engine_open()**: 
- Downloads image to DSP
- Releases DSP from reset
- DSP image initialization creates CE RMS
CODEC Engine: VIDENC_create()

```c
ARM (w Linux) User Space GUI
idevfd = open(...); // create
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);
while( doSvc ) { // execute
    read(idevfd, ...);
    VIDENC_control(myVE, ...);
    VIDENC_process(myVE, ...);
    write(ofilefd, ...); }
close(idevfd); // delete
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

**Code Interpretation**
- `VIDENC_create()` calls the codec engine driver.
- It creates a codec engine instance and a codec engine task (myVE TSK).
- It also activates a DSP task as an algorithm (xDM algo).

**Diagram Notes**
- CE RMS: Signals CE RMS to create algo instance.
- CE RMS creates TSK as algo’s context.
- Skeleton: unpacks args from IPC.

CODEC Engine: VIDENC_control()

```c
ARM (w Linux) User Space GUI
idevfd = open(...); // create
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);
while( doSvc ) { // execute
    read(idevfd, ...);
    VIDENC_control(myVE, ...);
    VIDENC_process(myVE, ...);
    write(ofilefd, ...); }
close(idevfd); // delete
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

**Code Interpretation**
- `VIDENC_control()` signals myVE TSK with algorithm context.
- it calls DSP VIDENC_control.

**Diagram Notes**
- CE RMS: Signals myVE TSK via RPC.
- myVE TSK calls DSP VIDENC_control.
CODEC Engine Details

CODEC Engine: VIDENC_process()

ARM (w Linux) User Space GUI
idevfd = open(...); // create
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);
while( doSvc )
  // execute
  read(idevfd, ...);
  VIDENC_control(myVE, ...);
  VIDENC_process(myVE, ...);
  write(ofilefd, ...);
close(idevfd); // delete
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);

Kernel Space Input Driver I Buf Output Driver O Buf

VISA API CODEC engine Stub

(*,*) xDM algo (local) I Buf O Buf

CODEC Engine: VIDENC_delete()

ARM (w Linux) User Space GUI
idevfd = open(...); // create
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);
while( doSvc )
  // execute
  read(idevfd, ...);
  VIDENC_control(myVE, ...);
  VIDENC_process(myVE, ...);
  write(ofilefd, ...);

Kernel Space Input Driver I Buf Output Driver O Buf

VISA API CODEC engine Stub

CE RMS VIDENC_delete Skeleton

myVE TSK

myVE DSP instance

(*,*) xDM algo (local) I Buf O Buf

CODEC Engine Details

2 - 24 DaVinci / OMAP Software Design Workshop - Introduction to DaVinci / OMAP Software
CODEC Engine: Engine_close()

```
CODEC Engine Details

idevfd = open(...); // create
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);
while( doSvc ){ // execute
    read(idevfd, ...);
    VIDENC_control(myVE, ...);
    VIDENC_process(myVE, ...);
    write(ofilefd, ...); }
close(idevfd); // delete
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```
Software Summary

DaVinci Technology Software Architecture

- **Application Layer (APL)**
  - User Space
    - GUI
    - Codec engine
    - Master thread
    - Mux/Demux
    - Network RTP/RTSP
- **Signal Processing Layer (SPL)**
  - CODEC engine
  - Video CODEC API
  - Imaging CODEC API
  - Speech CODEC API
  - Audio CODEC API

- **User Components**
  - AV Sync
  - Mux/Demux TS / ASF
  - Network RTP/RTSP

- **Available Drivers**
  - Video
  - Audio
  - UART
  - EMAC
  - I2C
  - SPI
  - USB 2.0
  - MMC/SD
  - 2 W’dogs
  - 2 Timers

- **I/O Buffers**
  - Shared Memory

---

DaVinci Technology Software Architecture: On TMS320DM6446 Processor

- **ARM**
  - Application Layer (APL)
  - User Space
  - Driver API
  - Kernel Space
    - Video
    - Audio
    - UART
    - EMAC
    - I2C
    - SPI
    - USB 2.0
    - MMC/SD
    - 2 Timers
    - 2 W’dogs

- **DSP**
  - Signal Processing Layer (SPL)
  - Codec engine resource server
  - DSP Link
  - DSP/BIOS

---

Software Summary
**APL Programmer View of DaVinci Technology System**

- **ARM**
  - Application Layer (APL)
  - GUI
  - Linux DRV API
  - VISA API
- **User Space**
  - Master thread
  - Multiplexer/Demultiplexer (Mux/Demux)
  - AV Sync
  - Network RTP/RTSP

**DaVinci Technology Strategy: Reduced Development Time**

- **Typical Video System Development Time**
  - OS & Device Drivers: 4 - 6
  - Frameworks & API: 6 - 10
  - CODECs: 4 - 16
  - HW Components: ~12
  - HW & SW Development Tools: ~6

- **DaVinci™ Technology System Development Time**
  - ~2
  - ~2
  - ~1
  - ~6
  - ~1
  - ~12 months

Duration (Months):
- Typically 32 – 50 months
- DaVinci™ Technology: ~12 months
For More Information

For More Information

- Codec Engine Server Integrator's User's Guide  (sprued5b.pdf)
- Prog. Details of Codec Engine for DaVinci Tech.  (spry091.pdf)
- Multimedia Frameworks Products (MFP) Product Folder:
  http://focus.ti.com/docs/toolsw/folders/print/tmdmfp.html
Introduction

This chapter begins with a very brief introduction to Linux. While we hope that most of you already are familiar with some Linux basics, we’ll spend a few minutes covering its high points.

In the second part of the chapter, we’ll review the VMware program and how we can use it to create a virtual Linux PC inside our Windows environment. If you can dedicate a PC to be used solely for Linux, that’s great. For many of us, who don’t have that option, VMware easily lets us use one – inside the other.

Finally, we end this chapter with a brief introduction to Das U-Boot, the Linux bootloader used to bootstrap TI’s Linux-based devices. Its ability to be configured through RS-232 makes it a handy choice. You’ll further explore some of the U-Boot config options in the lab accompanying this chapter.

Chapter Topics

Getting Started with the Tools ................................................................................................................. 3-1

Introduction to Linux .............................................................................................................................. 3-2
 Linux Fundamentals ................................................................................................................................. 3-2
 Basic Linux Commands – You should already know ........................................................................... 3-5
 VMware - Linux box in a Windows PC ............................................................................................. 3-6
 Das U-Boot ............................................................................................................................................3-12
 Sidebar: Getting to U-boot ....................................................................................................................3-13
 Using U-boot to start Linux ...................................................................................................................3-14
 U-Boot Variables & Commands.................................................................................................................3-16
 Boot Variations.........................................................................................................................................3-17
 Static vs. Dynamic Boot ........................................................................................................................3-21
 Configure U-Boot with Tera Term Macro’s ..........................................................................................3-22
 For More Information..........................................................................................................................3-22

Lab 3 – Experimenting with the DVEVM and U-Boot ...........................................................................3-23
 A. Verify Hardware Configuration...........................................................................................................3-24
 B. Test DVEVM Serial Interface...........................................................................................................3-24
 C. Start and Configure VMware and Ubuntu Linux...........................................................................3-25
 D. Configure the Boot Mode Options on the DVEVM .......................................................................3-29
 E. (Optional) Try other Options (Boot, VMware, etc.).......................................................................3-32
Introduction to Linux

Linux Fundamentals

1. **Bootloader**
   - Provides rudimentary h/w init
   - Calls Linux kernel and passes boot arguments

2. **Kernel**
   - Initializes the system (and device)
   - Manages system resources
   - Provides services for user programs

3. **Filesystem**
   - Single filesystem (/ root)
   - Stores all system files
   - After init, kernel looks to filesystem for “what’s next”
   - bootarg tells linux where to find root filesystem

Linux in Three Parts

---

*daVinci DM644x Workshop - Getting Started with the Tools*
Steps in Booting Linux

Linux Boot Process

Power On

<table>
<thead>
<tr>
<th>Boot Loader</th>
<th>U-Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Kernel</td>
<td>Initialize hardware</td>
</tr>
</tbody>
</table>

File System

<table>
<thead>
<tr>
<th>Init Process</th>
<th>/sbin/init – 1st process exe by kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login Prompt</td>
<td></td>
</tr>
</tbody>
</table>

Linux Kernel

What’s in the Linux Kernel

User Mode

- Custom User Application
- Gstreamer Framework
- 3D Graphics Library
- HTTP Server

Kernel Mode

- Process Management
- Memory Management
- Virtual File System
- Network Stack
- Device Management

- CPU support code
- CPU/MMU support code
- Storage device drivers
- Network device drivers
- Character device drivers

- CPU
- RAM
- MMC
- EMAC
- UART
Linux Root Filesystem

Red Hat : Root File System

Some folders common to Linux:

/dev – Common location to list all device drivers

/home - Storage for user’s files
  - Each user gets their own folder (e.g. /home/user)
  - Similar to “My Documents” in Windows
  - DVS DK GSG directory for TI tools, examples, working directory
  - "root" user is different, that user’s folder is at /root

/media – Usually find CDROM drive(s) mounted here

/mnt – Common location to mount other file systems
  - Linux only allows one filesystem
  - Add other disks (physical, network, etc) by mounting them to an empty
    directory in the root filesystem
  - Windows adds new filesystems (C:, D:, etc.) rather than using a single one

,opt – Common location for programs
  - MontaVista install MVLinux to /opt
  - akin to Windows “C:\Program Files”

/usr – Storage for user binaries
  - X86 Compiler for Red Hat programs (gcc) is stored in here

Filesystems: Red Hat vs. Montavista

Red Hat (PC) MontaVista (DVEVM)

- Tools/Host filesystem: Location our dev’l tools
- Target filesystem: filesystem to run on TI processors
- Notice the similarities between the two different Linux filesystems
- When releasing to production, it’s common to further reduce the target filesystem to eliminate cost
Basic Linux Commands – You should already know

Linux Command Summary

Some commands used in this workshop:

<table>
<thead>
<tr>
<th>File Management</th>
<th>Program Control</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ls</code> and <code>ls -la</code></td>
<td><code>&lt;ctrl&gt;-c</code></td>
</tr>
<tr>
<td><code>cd</code></td>
<td><code>ps</code>, <code>top</code></td>
</tr>
<tr>
<td><code>cp</code></td>
<td><code>kill</code></td>
</tr>
<tr>
<td><code>mv</code></td>
<td><code>renice</code></td>
</tr>
<tr>
<td><code>rm</code></td>
<td></td>
</tr>
<tr>
<td><code>pwd</code></td>
<td></td>
</tr>
<tr>
<td><code>tar</code> (create, extract .tar and .tar.gz files)</td>
<td><code>insmod</code>, <code>rmmod</code></td>
</tr>
<tr>
<td><code>chmod</code></td>
<td></td>
</tr>
<tr>
<td><code>chown</code></td>
<td></td>
</tr>
<tr>
<td><code>mkdir</code></td>
<td></td>
</tr>
<tr>
<td><code>mount</code>, <code>umount</code> (in general, what is “mounting” and how do you do it?)</td>
<td></td>
</tr>
<tr>
<td><code>alias</code></td>
<td></td>
</tr>
<tr>
<td><code>touch</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/sbin/ifconfig</code>, <code>ifup</code>, <code>ifdown</code></td>
<td></td>
</tr>
<tr>
<td><code>ping</code></td>
<td></td>
</tr>
<tr>
<td><code>nfs</code> (What is it? How to share a folder via NFS. Mounting via NFS.)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VMware Shared Folders</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/mnt/hgfs/&lt;shared name&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux Users</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>root</code></td>
<td></td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>su</code> (… exit)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BASH</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>What is BASH scripting</td>
<td></td>
</tr>
<tr>
<td>What are environment variables</td>
<td></td>
</tr>
<tr>
<td>How to set the PATH environment variable</td>
<td></td>
</tr>
<tr>
<td>What is .bashrc? (like DOS autoexec.bat)</td>
<td></td>
</tr>
<tr>
<td>man pages</td>
<td></td>
</tr>
<tr>
<td>change command line prompt</td>
<td></td>
</tr>
</tbody>
</table>

In this course, we will try to provide the necessary Linux commands needed to run the tools and such. Even so, if there are commands listed here that you don’t recognize, we recommend that you study up on them. To this end, the “Linux Pocket Guide” reference from the “0. Welcome Chapter” might be handy.
VMware - Linux box in a Windows PC

VMware – Run Linux inside Windows Application
VMware – Virtual Machine

Why VMware?
- Allows simultaneous use of Linux and Windows with one PC
- Virtual Linux PC within a Windows application
- VMware provides free “player” version of their software
- Virtual PC settings and hard disc image are stored inside any Windows folder
- Easily distribute virtual Linux PC with all DaVinci tools pre-installed
- By keeping known “good” copy of VMware image, you can easily reset Linux PC

Workshop VMware Image

Workshop VMware Image
- Ubuntu 10.04:
  - id = user, psw = useruser
- Notes:
  - Screensaver & Firewall off
  - NFS, TFTP, GIMP installed
  - VMware toolbox installed
- Installed DaVinci software following DVSDK “Getting Started Guide”
  - MV Linux 5.0 (Linux tools)
  - DaVinci LSP (kernel)
  - Target Filesystem
VMware – Free Player vs. Full Workstation

**Full Workstation**
- Can build VMware PC image from scratch
- “Snapshot” feature allows you to save & restore previous machine states (handy!)
- “Shared Folders” feature makes it easy to share files between Linux and Windows
- Not free, but small discount with current users referral code
- Workstation users get both full/free

**Free Player**
- Free
- Someone else has to create original VMware image (and do h/w mods)
- No snapshot feature

---

**VMware : Snapshot**

**Actions**
- Create new
- Delete
- Go To

**Deleting snaps**
- Deleting last one sets you back to previous state
- Delete middle one combines snapshots

**Performance**
- Too many snapshots can diminish perf.
- Too few and you could get stuck :-(

---

**Workshop VMware Image**
(Old snapshot example, latest image uses Ubuntu & MV5)
- Red Hat Enterprise 4 (RHEL4)
- Installed MV Linux 5 (demo version)
- Installed DVEVM
  “ARM development software”
- Installed DVSDK
  “DSP development software”
VMware : Shared Folders

Sharing folders
- VMware shared folders
- NFS
- Samba

VMware Shared Folders
- Easiest method
- Access from: /mnt/hgfs/shared
**VMware Filesystem**

Some important points in the preceding filesystem:

1. The directory structure on the left shows the root filesystem of the VMware image filesystem.

2. The first callout shows the `/home/user` folder. It contains a few interesting things:
   - `/home/user/dvevm_2_xx_xx` contains all the TI DVEVM and DVSDK content (cgt, bios, xdc, codec engine, examples, etc.). We placed this content here based on the DVEVM and DVSDK Getting Started Guides (GSG) instructions.
   - `/home/workdir` – a working directory, into it we made a copy of the Linux Support Package (LSP) and the default DM644x target root filesystem created by MV for the DVEVM.
   - `/home/user/workshop` and `/home/user/solutions` are the workshop lab folders. These directories are not shown on this screen capture.
3. The lower callout highlights the MontaVista tools in the /opt folder. In it:

   - /opt/montavista – contains the full version of MV Linux. The full version is sold by MontaVista.

   - /opt/mv_pro_5.0/montavista – contains the **demo** version of MV Linux which ships with the DVSDK. It was placed in a different folder (1 level down) to keep it from conflicting with the full version of MV Linux. *We installed this as part of the workshop VMware image.*

   If you install both the MontaVista licensed software, and all the DVSDK software, you may end up with both of these in your filesystem (as shown above). *For our workshop VMware image, though, we only installed the demo version at /opt/mv_pro_5.0/montavista.*

4. Finally, in the next topic we discuss booting the DVEVM. For Linux to boot, the bootloader (Das U-Boot) needs to find the Linux kernel; this is usually found either in the flash memory on the board, or the bootloader can download it from a network. When the latter is chosen, the default is to TFTP the kernel (“uImage” file) from the /tftpboot folder.
Das U-Boot

The DaVinci EVM board is delivered with the open-source boot loader, Das U-Boot (U-Boot).

U-Boot resides in NOR flash memory, copies itself to DDR, and executes itself from DDR when the DaVinci EVM is powered on.

In general, U-Boot performs the following functions:
1. Initializes the DaVinci EVM hardware
2. Provides boot parameters to the Linux kernel
3. Starts the Linux kernel

Additional U-Boot Features

In addition, it provides some convenient features that help during development:
1. Ping IP addresses
2. Reads and writes arbitrary memory locations
3. Uploads new binary images to memory via serial, or Ethernet
4. Copies binary images from one location in memory to another

U-Booting from NAND Flash

The DVEVM can be booted from NAND flash
The NAND boot process is a bit more involved
By default, the DVEVM boots from NOR flash
For more information on setting up NAND boot, please refer to:

Booting and Flashing via the DaVinci TMS320DM644x Serial Interface (spraai4.pdf)
Basic Application Loading over the Serial Interface for the DaVinci TMS320DM644x (spraai0.pdf)
Sidebar: Getting to U-boot

Bootloader Components

<table>
<thead>
<tr>
<th>Boot stage</th>
<th>Operations</th>
<th>User Config’d</th>
<th>DaVinci</th>
<th>OMAP3x</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-level</td>
<td>This is ROM’d code for detecting desired boot type (NAND, UART, …) and loading executable code of second-level bootloader from selected peripheral/interface</td>
<td>No</td>
<td>RBL</td>
<td>RBL</td>
</tr>
<tr>
<td>Second-level</td>
<td>The primary function of this boot loader is to initialize external memory and system clocks so that a larger, more advanced boot loader (in this case U-boot) can be loaded.</td>
<td>Board Designer</td>
<td>UBL</td>
<td>XLDR</td>
</tr>
<tr>
<td>Linux boot</td>
<td>“Das U-boot” is the standard open-source Linux boot loader for ARM. It supports networking for TFTP/NFS booting. It is used to locate, load and execute the Linux kernel in uImage format and is also responsible for passing arguments to the kernel.</td>
<td>Yes</td>
<td>U-boot</td>
<td>U-Boot</td>
</tr>
</tbody>
</table>

Customizing UBL / XLDR
1. Configure system clocks
2. Setup memory interfaces

* In this workshop we will only configure the 3rd level bootloader (Das U-boot).
Using U-boot to start Linux

To Boot Linux, You Need...

1. Bootloader (U-Boot)
   - At reset, U-Boot bootloader is executed
2. Linux Kernel
   - U-Boot loads O/S kernel into DDR2 memory; then,
   - Connects to the root filesystem
     If you don’t know what this is, think of it as the ‘c:\’ drive of in Windows PC
3. Filesystem

Where Do You Find …

<table>
<thead>
<tr>
<th>Where located:</th>
<th>DVEVM Board (default)</th>
<th>After Lab 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Bootloader (U-Boot)</td>
<td>Flash</td>
<td>Flash</td>
</tr>
<tr>
<td>2. Linux Kernel</td>
<td>Flash</td>
<td>Linux PC (via TFTP)</td>
</tr>
<tr>
<td>3. Filesystem</td>
<td>DVEVM Hard Drive</td>
<td>Linux PC (via NFS)</td>
</tr>
</tbody>
</table>

“HDD boot” “NFS boot”

- By default, the DVEVM ships in “HDD boot” mode; this allows the demo applications to run "out-of-the-box"
- “NFS boot” (network boot) is good for application development
Using U-boot to start Linux

**NFS Boot**

- RS-232 is physical connection to U-Boot
- Use to stop DVEVM from stand-alone boot
- Configure U-Boot's modes by setting/saving environment variables

- Ethernet provides physical connection to Linux PC
- Root filesystem is accessed via NFS protocol
- Don't need to 'flash' DVEVM after compiling new program

Note: ~/workdir/filesys = /home/user/workdir/filesys

**Configuring U-Boot and Starting Linux (5 Steps)**

1. Connect an RS232 serial cable and start a Tera Term
2. Power on the DVEVM and press any key in TeraTerm to abort the boot sequence
3. Set U-Boot variables to select how Linux will boot (save changes to flash to retain settings after power cycle)
4. Boot Linux using either:
   - the U-Boot “boot” command
   - power-cycle the DVEVM
5. After Linux boots, log in to the DVEVM target as “root”
   - Note, login with: “user” for the Tools Linux PC
     “root” for the DVEVM target
   - You can use any RS-232 comm application (Linux or Win), we use Tera Term for its macro capability
U-Boot Variables & Commands

Configuring U-Boot

Common Uboot Commands:
- printenv - prints one or more uboot variables
- setenv - sets a uboot variable
- saveenv - save uboot variable(s)
- run - evaluate a uboot variable expression
- ping - (debug) use to see if Uboot can access NFS server

Common Uboot Variables:
- You can create whatever variables you want, though some are defined either by Linux or common style
  - bootcmd - where Linux kernel should boot from
  - bootargs - string passed when booting Linux kernel
    e.g. tells Linux where to find the root filesystem
  - serverip - IP address of root file system for NFS boot
  - nfspath - Location on serverip for root filesystem

U-Boot Macros

- Variables can reference each other
  for example, to keep original bootargs settings, try:
    setenv bootargs_original $(bootargs)

- “Run” command – Force uboot to evaluate expressions using run
  For example, evaluate this expression:
    setenv set_my_server 'setenv $(serverip):$(nfspath)'
  Using the run command:
    setenv serverip 192.168.1.40
    setenv nfspath /home/user/workdir/filesys
    run set_my_server
Using U-boot to start Linux

U-Boot Commands/Variables

| U-Boot Commands: setenv, saveenv, printenv, run, ping, dhcp, tftp |
| Example: setenv ipaddr 192.168.1.41 |

Common variables used to configure the behaviour of U-Boot:

- **autoload**: If set to “no”, only lookup performed, no TFTP boot
- **autostart**: If “yes”, a loaded image is started automatically
- **baudrate**: Baudrate of the terminal connection, defaults to 115200
- **bootargs**: Passed to the Linux kernel as boot “command line” arguments
- **bootcmd**: Command string that is executed when the initial countdown is not interrupted
- **bootdelay**: After reset, U-Boot waits ‘bootdelay’ seconds before executing bootcmd; abort with any keypress
- **bootfile**: Name of the default image to load with TFTP
- **cpuclock**: Available for processors with adjustable clock speeds
- **ethaddr**: Ethernet MAC address for first/only ethernet interface (eth0 in Linux); additional ethernet if’s use eth1addr, eth2addr, ...
- **initrd_high**: Used to restrict positioning of initrd ramdisk images
- **ipaddr**: IP address; needed for tftp command
- **loadaddr**: Default load address for commands like tftp or loads
- **loads_echo**: If 1, all characters recv’d during a serial download are echoed back
- **pram**: Size (kB) to reserve for “Protected RAM” if the pRAM feature is enabled
- **serverip**: TFTP server IP address; needed for tftp command; (also use for nfs root mnt’s)
- **serial#**: Hardware identification information, usually during manufacturing of the board
- **silent**: If option is enabled for your board, setting this will suppress all console msgs

Boot Variations

<table>
<thead>
<tr>
<th>Mode</th>
<th>IP</th>
<th>Linux Kernel</th>
<th>Root Filesystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dhcp</td>
<td>Flash</td>
<td>HDD</td>
</tr>
<tr>
<td>2.</td>
<td>dhcp</td>
<td>Flash</td>
<td>NFS</td>
</tr>
<tr>
<td>3.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>HDD</td>
</tr>
<tr>
<td>4.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>NFS</td>
</tr>
</tbody>
</table>

U-Boot's bootcmd variable specifies where to find the kernel used to boot Linux

Flash: setenv bootcmd bootm 0x2050000

TFTP: setenv bootcmd 'dhcp;bootm'
Boot Variations

<table>
<thead>
<tr>
<th>Mode</th>
<th>IP</th>
<th>Linux Kernel</th>
<th>Root Filesystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dhcp</td>
<td>Flash</td>
<td>HDD</td>
</tr>
<tr>
<td>2.</td>
<td>dhcp</td>
<td>Flash</td>
<td>NFS</td>
</tr>
<tr>
<td>3.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>HDD</td>
</tr>
<tr>
<td>4.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>NFS</td>
</tr>
</tbody>
</table>

U-Boot’s bootargs variable specifies where to find the root filesystem

- **HDD**
  - `setenv bootargs` `console=ttys0,115200n8` `noinitrd rw ip=dhcp root=/dev/hda1, nolock` `mem=120M`

- **NFS**
  - `setenv bootargs` `console=ttys0,115200n8` `noinitrd rw ip=dhcp root=/dev/nfs nfsroot=$(serverip):$(nfspath),nolock` `mem=120M`

Configuring U-Boot

Kernel from **Flash**, Filesystem from **HDD**

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttys0,115200n8
                     noinitrd rw ip=dhcp root=/dev/hda1 nolock
                     mem=120M
[rs232]# saveenv
```

Kernel from **Flash**

Filesystem from **HDD**
Configuring U-Boot
Kernel from **Flash**, Filesystem from **NFS** (network)

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x205000
[rs232]# setenv bootargs console=ttyS0,115200n8
  noinitrd rw ip=dhcp root=/dev/nfs
  nfsroot=$(serverip):$(nfspath),nolock
  mem=120M
[rs232]# saveenv
```

Configuring U-Boot
Kernel via **TFTP**, Filesystem from **NFS** (network)

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd 'dhcp;bootm'
[rs232]# setenv bootargs console=ttyS0,115200n8
  noinitrd rw ip=dhcp root=/dev/nfs
  nfsroot=$(serverip):$(nfspath),nolock
  mem=120M
[rs232]# saveenv
```
Save Time by using U-Boot Macros

► Add these commands to your U-Boot setup to quickly switch between HDD and NFS boot configurations.
► The next slide shows how to use these macros.

[evm]# setenv bootargs_hdd 'setenv bootargs console=ttyS0,115200n8
noinitrd rw ip=dhcp root=/dev/hda1, nolock mem=120M'
[evm]# setenv bootargs_nfs 'setenv bootargs console=ttyS0,115200n8
noinitrd ip=dhcp root=/dev/nfs nfsroot=$(serverip):$(nfspath)
mem=120M'
[evm]# saveenv

Configuring U-Boot Using Macro’s

HDD boot:
[rs232]# run bootargs_hdd
[rs232]# saveenv
[rs232]# run bootcmd

NFS boot:
[rs232]# setenv serverip 192.168.1.103
[rs232]# saveenv nfspah /home/user/workdir/filesys
[rs232]# run bootargs_nfs
[rs232]# saveenv
[rs232]# run bootcmd
Static vs. Dynamic Boot

### Booting with Static IP Addresses

<table>
<thead>
<tr>
<th>Mode</th>
<th>IP</th>
<th>Linux Kernel</th>
<th>Root Filesystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dhcp</td>
<td>Flash</td>
<td>HDD</td>
</tr>
<tr>
<td>2.</td>
<td>dhcp</td>
<td>Flash</td>
<td>NFS</td>
</tr>
<tr>
<td>3.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>HDD</td>
</tr>
<tr>
<td>4.</td>
<td>dhcp</td>
<td>TFTP</td>
<td>NFS</td>
</tr>
<tr>
<td>5.</td>
<td>static</td>
<td>Flash</td>
<td>HDD</td>
</tr>
<tr>
<td>6.</td>
<td>static</td>
<td>Flash</td>
<td>NFS</td>
</tr>
<tr>
<td>7.</td>
<td>static</td>
<td>TFTP</td>
<td>HDD</td>
</tr>
<tr>
<td>8.</td>
<td>static</td>
<td>TFTP</td>
<td>NFS</td>
</tr>
</tbody>
</table>

**U-Booting : Static vs Dynamic IP**

- You must specify the IP addresses
- Everywhere we previously had `dhcp` must now reference the static ip addresses
- This example creates a variable called `myip` and used it in place of the previous `dhcp` entries in `bootargs`

```bash
[rs232]# setenv serverip 192.168.13.120
[rs232]# setenv ipaddr 192.168.13.121
[rs232]# setenv gateway 192.168.13.97
[rs232]# setenv netmask 255.255.255.224
[rs232]# setenv dns1 156.117.126.7
[rs232]# setenv dns2 157.170.1.5
[rs232]# setenv myip $(ipaddr):$(gateway):$(netmask):$(dns1):$(dns2)::off
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttyS0,115200n8 nolinux rw
   ip=$(myip) root=/dev/nfs nfsroot=$(serverip):$(nfspath),nolock mem=120M $(videocfg)
[rs232]# saveenv
```
Configure U-Boot with Tera Term Macro’s

Using Tera Term Macros

- U-boot strings are very precise, one wrong character will prevent booting
- TeraTerm Pro (freeware) provides a simple macro language
- We have included a script which inquires how you want to boot, then sets the uboot variables accordingly
- Macro name is: `tto_uboot_setup.ttl`

For More Information

- The Linux Filesystem Explained
  
  http://www.freeos.com/articles/3102/
Lab 3 – Experimenting with the DVEVM 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:

A: Verify hardware configuration  
B: Test DVEVM serial interface  
C: Start and configure Ubuntu Linux under VMware  
D: Configure the boot mode options on the DVEVM  
E: (Optional) Experiment with Shared folders, VM Snapshots, and other boot options
A. Verify Hardware Configuration

1. **Power off the DM6446 DVEVM.**
2. **Verify** that S3 switch on the DaVinci DVEVM Board is set to:
   - NTSC: 10111 11110
   - PAL: 10111 11111
3. **Connect** an Ethernet cable between the DVEVM and the PC’s Ethernet adaptor.
4. **Connect** an RS-232 cable between the DVEVM and PC RS-232 ports.

B. Test DVEVM Serial Interface

5. **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

6. **Start the DVEVM:** Power on the DVEVM board and **press any key** to interrupt U-Boot’s boot sequence. At this point, the DVEVM U-Boot terminal prompt DaVinci EVM # should be visible in the TeraTerm session window.

7. **Verify the DVEVM Linux kernel:** At the DaVinci EVM # prompt in the TeraTerm window type `imls`. The first line should state: **Image at 02050000**; if not, ask the instructor for assistance.

8. **Note initial DVEVM environment status:** Again, at the DaVinci EVM # prompt, type `printenv`. Note briefly the size of the list and scan their values. These will change considerably later in this procedure.

9. **Note the Windows Ethernet addresses:** This information will be needed to test the Linux Ethernet connection in the next section of the lab, and is easiest to access now before starting Linux. 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 writing these values.

   **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
C. Start and Configure VMware and Ubuntu Linux

10. Launch VMware: On the Windows desktop, **double click** the **VMware** icon:

11. Open the TTO workshop VMware image: In the VMware Workstation window, **Home tab**, **Click** on the **Open Existing VM or Team** Icon:
From the dialog box that appears, go to directory **C:vm_images** and select folder **tto_vm_image_(v2.10)** (version was v2.10 at this writing) and choose file **tto_vm_image_v2.10.vmx**

**Note:** In USA classrooms, the VMware image is broken into two parts, the ‘child’ image (~300MB) is specified in this step which depends upon a ‘parent’ image (~16GB) in order to work. The parent image is found in:

E:\vm_parent\TTO_vm_parent_(v2.10)\

Breaking the image into two parts allows us to re-image the C:\ drive without needing to reload the entire 16GB for each class.

12. 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:

![Ethernet Setting](image)

If you happened to get a little ahead of our instructions and already started the VMware image (which we do in step 15), the easiest way to see this is in the status bar. Just hover over the Ethernet board icon and read the popup message:

![Ethernet: Bridged](image)

**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.
13. 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 Settings... 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:

14. 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.)
15. **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.)

16. **Log in to Linux:** When Ubuntu reaches the login screen, log in as follows:

   **Userid:** user  
   **Password:** useruser

17. **Open a terminal window.**

   ![Terminal window](image)

   The easiest way to open the terminal is to click its 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.

18. **Determine the Ubuntu Ethernet address:** In the terminal window, run `ifconfig` by typing:

    ```
    /sbin/ifconfig eth0
    ```

    and transcribe the IP address below.

   **IP Address** _______________________ (note, it will be called “inet” in the Linux response)

---

**Sidebar**

We have placed the Ubuntu Network Manager icon in the notifications area of the Ubuntu toolbar panel. Sometimes the icons are shuffled a bit by Ubuntu, but it should look something like this:

By clicking on the Network Manager icon, you will see the two connection states we created for our Ubuntu image.

- **Static_IP** (IP address set statically to 192.168.1.40)
- **Dynamic_IP** (IP address assigned by your networks DHCP controller)

In TTO USA classes, you should be using the Static_IP configuration, as we use this to connect directly to the board from the PC without using an Ethernet router.
19. *Test the Linux Ethernet port:* “Ping” the Windows Ethernet port to verify the operational status of the Linux port. From the terminal window type **ping <IP Address>** where “IP address” is the value recorded in step B9 above. If successful, you should observe a display like the one below. Press **<Ctrl> C** to halt the pinging.

```
[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
```

At this time, we should have the following IP addresses set:

- **Windows PC:** 192.168.1.39
- **Ubuntu Linux:** 192.168.1.40
- **DVEVM target:** 192.168.1.41
D. Configure the Boot Mode Options on the DVEVM

19. *Return to Windows and TeraTerm:* Since VMware creates a complete virtual PC when the cursor is within its borders, it is necessary to place 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. 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.

20. *Test network connection from DVEVM to Ubuntu VM image:*

   This is a good thing to check, since we plan to boot the DVEVM across the network – that is, we plan to get the Linux Kernel for the DVEVM, as well as the root filesystem from our Ubuntu Linux VM image.

   Run the **ping** command from Uboot:

   ```
   ping 192.168.1.40
   ```

   It should respond with: **Connection is alive**

21. *Run the TeraTerm macro to setup the DVEVM U-Boot mode:* From TeraTerm, select **Control | Macro.** From directory **C:\Program Files\TTERMPRO** select file **tto_uboot_setup.ttl.** As the macro runs, make the following selections:

   - Video mode? **[Use NTSC or PAL]**  
     Yes (for NTSC)
   - Use Default NFS Server IP Address: **192.168.1.40**  
     Yes
   - Boot Static or Dynamic? **[Yes=Static, No=Dhcp]**  
     Yes
   - Root filesystem from NFS or HDD? **[Yes=NFS, No=HDD]**  
     Yes
   - Boot Kernel via TFTP or from Flash? **[Yes=TFTP, No=Flash]**  
     Yes
   - Save bootargs?  
     Yes
   - Boot Linux now? (No, we’ll do this manually)  
     No

22. Examine the DVEVM’s Linux environment: As before, type **printenv** and compare to what was observed previously. **Note: changes to the environment must be saved to the Flash to remain active after restarting the DVEVM hardware, which was done automatically by the macro when you answer “Yes” to the ‘save bootargs’ question.**

   To manually preserve the bootargs, type:

   ```
   save
   ```
23. *Optional:* 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.

24. *Reboot:* power-cycle the DVEVM or type `boot` to restart the DVEVM with the new
environment selections. When bootup completes (in Tera Term), log in as **root user** (no
password 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 DVEVM 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 DVEVM using the root user.
In embedded applications, there often only exists a single user (root).
25. **Verify shared file system between Ubuntu and DVEVM.** Since any file change to the root directory of our DVEVM 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 DaVinci board):

```
root@192:~# cd /
```

 moves you to root

```
root@192:~# 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/workdir/filesys
[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 filesys folder, they’re immediately available at our NFS mounted DVEVM target.

**Review**

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

<table>
<thead>
<tr>
<th>DaVinci Board “Target”</th>
<th>Ubuntu Linux “VMware Image”</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

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?  

__________________________________________________________________________
E. (Optional) Try other Options (Boot, VMware, etc.)

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. (Probably won’t work with if you are working these lab exercises on a notebook computer.)

b. Read and write a file to/from Windows using VMware’s shared folders feature:

Windows: C:\vm_images\shared
Linux: /mnt/hgfs/shared

c. Try creating a VMware snapshot.
Chapter Topics

Tools Overview ........................................................................................................................................ 4-1

Development Platforms .......................................................................................................................... 4-2
  Hardware ............................................................................................................................................. 4-2
  Software Development Kits (SDK, DVSDK) ...................................................................................... 4-5

Software Developer Roles ..................................................................................................................... 4-7

Linux Tools .......................................................................................................................................... 4-8
  Linux Tools Summary ......................................................................................................................... 4-8
  IDE Examples ................................................................................................................................... 4-9

DSP Tools ........................................................................................................................................ 4-11

Linux Distributions ............................................................................................................................... 4-12
  What are distributions? ...................................................................................................................... 4-12
  O/S Choices ..................................................................................................................................... 4-14
  Community Options .......................................................................................................................... 4-15
  Commercial Options .......................................................................................................................... 4-18

(Optional) TI SDK : Future Directions ................................................................................................. 4-20

(Optional) How Do I Rebuild Linux ... and Why? ................................................................................. 4-21
Development Platforms

Hardware

DM6446 DVEVM

DM365 DVEVM

- NAND
- DDR
- SD/MMC x 2
- Power
- UART
- JTAG Plug
- 10/100 RJ45
- Imager I/F
- Audio IO
- Video Out I/F
- Video In I/F
- Cursor Pad
Modular EVM Kits – AM3517 Example

SOM Module
AM3517 SOM-M2
Price: <$100
SW Development
1.6" x 2"
Features:
- 256 MB DDR2 SDRAM
- 512 MB NAND flash
- Wired Ethernet
- Wireless 802.11b/g/n*
- Bluetooth 2.1 + EDR IF*
- Self-boot Linux image
- Purchase – Logic via Arrow, Avnet, Digikey
- Support – Logic

eXperimenter Board
SOM

EVM
TMDXEVM3517
Price: $999
Full Development Platform
- EVM additionally includes:
  - LCD
  - Multimedia In/Out
  - KeyPad
  - Connect: CAN, RJ45, USB, UART, stacked SD
  - Channel – TI & distribution
  - Support – TI & Logic
  - Linux and WinCE SDK’s (from TI); Android SDK is in development

eXperimenter Kit
SDK-XAM3517-10-256512R
Price: $199
S/W and H/W Dev’t
5" x 6"
Features SOM features +
- HDMI (video only)
- MMC/SD card slot
- Network/USB/Serial/JTAG/Logic-LCD Connectors
- Built-in XDS100 emulation
- Purchase – Logic via Arrow, Avnet, Digikey
- Support – Logic
- SW: Linux, WinCE

SOM Module
AM3517 SOM-M2
Price: <$100
SW Development
1.6" x 2"
Features:
- 256 MB DDR2 SDRAM
- 512 MB NAND flash
- Wired Ethernet
- Wireless 802.11b/g/n*
- Bluetooth 2.1 + EDR IF*
- Self-boot Linux image
- Purchase – Logic via Arrow, Avnet, Digikey
- Support – Logic

4 Types of Hardware Development Tools

System-on-Module
Use Case
- Simplify system board design
- Medium for Prototype or Production end equipment

Community Board
Use Case
- Evaluation of processor functionality
- Application development with limited peripheral access
- NOT supported by TI

eXperimenter Kit
Use Case
- Evaluation of processor functionality
- Application development with limited peripheral access

Evaluation Module
Use Case
- Touch-screen application development with full peripheral access
- Application specific development

Hardware Development Environments

Texas Instruments
## Evaluation and Development Kits

### Development Kit Contents:
- Evaluation board and Documentation
- Software Development Kits
- Development Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Part Number</th>
<th>Price</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM37x EVM</td>
<td>TMDXEVM3715</td>
<td>$1495</td>
<td>TI / Mistral</td>
</tr>
<tr>
<td>AM/DM37x Eval Module</td>
<td>TMDX3730EVM</td>
<td>$1495</td>
<td>TI / Mistral</td>
</tr>
<tr>
<td>OMAP35x EVM</td>
<td>TMDSEVM3530</td>
<td>$1495</td>
<td>TI / Mistral</td>
</tr>
<tr>
<td>AM3517 EVM</td>
<td>TMDXEVM3517</td>
<td>$999</td>
<td>TI / Logic</td>
</tr>
<tr>
<td>AM18x EVM</td>
<td>TMDXEVM1808L</td>
<td>$1150</td>
<td>TI</td>
</tr>
<tr>
<td>OMAP-L138 EVM</td>
<td>TMDXOSKL138BET</td>
<td>$849</td>
<td>TI / Logic</td>
</tr>
<tr>
<td>AM17x EVM</td>
<td>TMDXEVM1707</td>
<td>$845</td>
<td>TI</td>
</tr>
<tr>
<td>AM18x Experimenter Kit</td>
<td>TMDXEXP1808L</td>
<td>$445</td>
<td>TI</td>
</tr>
</tbody>
</table>

## Community Boards & Modules

### To Access:
Contact TI partners for more information or click link to buy now

<table>
<thead>
<tr>
<th>Tool</th>
<th>Part Number</th>
<th>Price</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Cost Kits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beagle Board (OMAP35x)</td>
<td>Beagle</td>
<td>$149</td>
<td>Community</td>
</tr>
<tr>
<td>Hawkboard (OMAP-L138)</td>
<td>ISSPL-Hawk</td>
<td>$89</td>
<td>Community</td>
</tr>
<tr>
<td>SOM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMAP35x System on Module</td>
<td>OMAP35x SOM-LV</td>
<td>$99</td>
<td>Logic</td>
</tr>
<tr>
<td>Overo OMAP35x Computer on Module</td>
<td>Overo</td>
<td>$149-219</td>
<td>Gumstix</td>
</tr>
<tr>
<td>KBOC OMAP35x System on Module</td>
<td>KBOC</td>
<td>$139</td>
<td>KwikByte</td>
</tr>
</tbody>
</table>

* Prices subject to change
Software Development Kits (SDK, DVSDK)

<table>
<thead>
<tr>
<th>S/W Dev'l Kit</th>
<th>Description</th>
<th>Processor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux PSP SDK</td>
<td>Small Linux Distro supporting TI ARM devices</td>
<td>OMAP35, AM35, AM18, OMAP-L1, DM644x, DM6467, DM3xx</td>
</tr>
<tr>
<td>Graphics SDK</td>
<td>Graphix SVSGX development kit OPENGL ES / VG demos, drivers, targets, Getting Started Guide</td>
<td>OMAP3515, OMAP3530, AM3517</td>
</tr>
<tr>
<td>“DVSDK”</td>
<td>TI provided libraries, examples, demos Codec Engine (VISA), DSPlink, Codec/Algos (XDM), BIOS, XDC, Linux utilities, etc.</td>
<td>All TI SOC’s: ARM, DSP, ARM+DSP obviously, not all devices require all the s/w components</td>
</tr>
</tbody>
</table>

Code Gen Tools (not really “kits” per se)
- Linux GNU Compiler (CodeSourcery)
- C6000 DSP Compiler (TI)
- All TI ARM and DSP devices where appropriate

- PSP is a TI specific acronym that represents the name of the group inside of Texas Instruments which “owns” the kernel and driver development activities: Platform Support Package team
- Wireless SDK is available independently of these other kits to support the TI WLxxxx Bluetooth/WiFi devices

Wireless SDK: Getting started with WL1271 on OMAP35x EVM

Software
- Pre-integrated with TI’s SDK
- WLAN and Bluetooth® software support (FM support not included)
- Pre-tested against WiFi and Bluetooth® specifications
- Open Source Linux drivers
  - Kernel 2.6.x
  - TI WLAN driver
  - BlueZ Bluetooth® stack
- Windows® CE 6.0 drivers
  - Available in mid 2010
  - Microsoft WiFi and Bluetooth® stacks
  - Adeneo’s Bluetooth Manager
- User Guides
- Complete API reference
- Application Notes
- Demo applications and sample code

Hardware
- Wireless Connectivity Card
  - WL1271 module with integrated TCXO
  - 2.4GHz chip antenna (default configuration)
  - U.FL antenna connector (optional configuration)
  - Plugs into EVM’s Expansion Connector (supported on EVM Rev G)

Development tools and partners
- Compatible with EVM’s toolchain
- Wireless Connectivity Card reference schematics
- Command Line Interface (CLI) to configure and exercise WLAN & Bluetooth® applications
- Partners
  - LS Research: WL1271 module
  - Mistral: Linux System Integrator
  - Adeneo: WinCE Syst. Integrator

Linux-Based DaVinci Workshop - Tools Overview 4 - 5
## Code Generation Tools: CodeSourcery

- TI SDK’s recommend the CodeSourcery Lite version, although you may want to upgrade to a more complete solution.
- Check SDK/DVSDK or Linux PSP release notes for tool versions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Lite</th>
<th>Personal</th>
<th>Professional</th>
</tr>
</thead>
<tbody>
<tr>
<td>30-day Installation Support</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Unlimited Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Priority Defect Correction</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Access to Updates, Knowledge Base</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Big Endian, Neon support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GNU C/C++ Compiler</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GNU Debugger (GDB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Eclipse IDE</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Annual Subscription Price (per Host)</td>
<td>Free</td>
<td>$399</td>
<td>$2799</td>
</tr>
</tbody>
</table>

## What We Use In This Workshop: DM6446 DVSDK

### Software Components
- MontaVista Linux (Pro 5 demo)
- Linux-based Drivers (PSP)
- Codec Engine (framework)
- DSP/BIOS
- RTSC Packages / XDC Config Tool
- XDAIS/XDM Codec API

### Demo Software:
- Demos include provide you with insight into DaVinci technology capabilities:
  - Encoders (H.264, MPEG4)
  - Decoders (H.264, MPEG4, MPEG2)
  - Simultaneous encode/decode (H.264)
  - Video pass-through
  - Networking demo
- Stored on the HDD, ready to run right out of the box
- Simple graphical user interface controlled via IR remote (or from command line)
- Source code provided down to VISA layer
- Demos can be recompiled with provided CODEC combinations:
  - To create other Codec combinations, you must request free Codec eval’s
- Demos provide on-screen information regarding:
  - ARM & DSP loadings
  - HDD storage information (encode only demo)
Software Developer Roles

What Role Do You Play?

Linux Open Source Community Software

**Application Layer (APL)**
- **Application Programmer**
  - Uses all the resources of Linux to build/maintain app
  - Writes to VISA interface to harness the power of SPL

**Signal Processing Layer (SPL)**
- **DSP Author**
  - Creates signal processing algorithms (CODECs) to be initiated by APL
  - Standard APIs (xDM, xDAIS) provided so DSP author can write and test algorithms independent of calling application
  - The TI third party network has already created 1000’s of eXpressDSP compliant algo’s
  - TI and its ASPs have created xDM CODECs to address the needs of VISA applications

**I/O Layer (IOL)**
- **Driver Author**
  - TI authored many drivers; MV provides to open source
  - Driver author only needs to write to standard O/S APIs

**Kernel Maintainence**
- Rebuilds kernel with new/updated drivers (as req’d)
- Substantially easier with MontaVista license support

**User Space**

**Driver API**

**VISA API**

**Embedded System Developer - Creates DSP Executable**
- Builds DSP Server from Codecs (i.e. builds the DSP executable image)
- Manages DSP’s use of memory
## Linux Tools

This section contains information on the Linux tools used for these developer roles.

► Application Tools (ARM)
► Driver Development
► Kernel Maintainence

You’ll find subsections that address tools for these roles by their vendor.

### Linux Tools Summary

<table>
<thead>
<tr>
<th>ARM/Linux Tools</th>
<th>Application Author</th>
<th>Driver Development</th>
<th>Linux Kernel &amp; Filesystem</th>
</tr>
</thead>
</table>
| MontaVista (MVL) DevRocket               | *“Best in class”* for Linux application development                               | IDE debugger for driver code development                                          | *“Best in class”*
                                | - GNU ARM compiler (GCC)                                                             | KGDB based debug                                                                     | DevRocket’s Kernel & Platform (filesystem) projects make quick and easy for this category |
                                | - GDB-based debug                                                                  |                                                                                   |                                                                                          |
                                | - Eclipse-based IDE                                                                |                                                                                   |                                                                                          |
| Green Hills (GHS) Multi IDE              | *Also “Best in class”* for Linux application development                           | *“Best in class”* debug support for Linux driver development                      | Rudimentary support for building the Linux Kernel                                         |
                                | - Supports multiple OSs: INTEGRITY, Linux, etc.                                    | *IDE debugger for driver code development                                        |                                                                                          |
                                | - GHS optimized ARM compiler                                                        | *Advanced debug target server – use KGDB or JTAG to connect to target             |                                                                                          |
| Texas Instruments Code Composer          | Alpha/beta versions are free via the TI website                                    | IDE debugger for driver code development                                          |                                                                                          |
                                | (CCStudio v5 Alpha)                                                                | *JTAG to connect to target                                                        | Not applicable                                                                            |
| Open Source                              | GDB                                                                                | KGDB                                                                              | Standard command-line (or asci-based gui) support for building the linux kernel          |
IDE Examples

Eclipse : CCSv5, MV DevRocket, Code Sourcery

Code Composer Studio

- Integrated Edit / Debug GUI
- Code Generation Tools
- BIOS: Real-time kernel
  - Real-time analysis
Green Hills Software

Multi IDE: Simultaneous Dual-Core DaVinci™ Technology ARM/C64x+ Debugging

- Advanced target debug server
- Loading and debugging images on both cores
- Multi-core debug
  - Individual core instruction stepping
  - Processor context debugging and viewing (e.g. registers, variables, ...)
- Single instance of MULTI debugger
- OS aware debugging on all cores (Linux, INTEGRITY, DSP/BIOS™ kernel, ...)

Integrity RTOS

- Prioritized, preemptive, deterministic ARM O/S
  - Scheduler more like DSP/BIOS™ kernel than Linux-like
  - Not time sliced
  - Thread priorities determined by user only
  - Offers thread (task) and process (memory) switch ability
  - Multiple scheduler domain option
    - Allows N individual scheduler environments
    - Each can be assigned a minimum slice of time
    - Thread starvation avoidance mechanism
    - Unused slice times returned to other slices – no waste
- Build size in 200-300K range, depending on features employed
- Flight certified
- No royalties per unit – O/S rights purchased per project or company
- Good option in systems where Linux is not the ‘right fit’
DSP Tools

► Codecs (i.e. Algo) Development
► Creating DSP Executable

### DSP Development Tools

<table>
<thead>
<tr>
<th>DSP Tools</th>
<th>DSP Programmer</th>
<th>Embedded System Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>MontaVista DevRocket</td>
<td>• Not Applicable</td>
<td>• TI Code Generation tools provide system configuration and building of executable DSP embedded image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Multi-proc (ARM-DSP) support may aide with finding problems with final system integration</td>
</tr>
<tr>
<td>Green Hills (GHS) Multi IDE</td>
<td>• Single IDE for ARM and DSP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Supports ARM + DSP devices</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Only support basic DSP/BIOS visualization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• GHS uses Ti’s Optimizing DSP Compiler</td>
<td></td>
</tr>
<tr>
<td>Texas Instruments Code Composer Studio (CCStudio)</td>
<td>• Highly-integrated development environment</td>
<td>• TI Code Generation tools provide system configuration and building of executable DSP embedded image</td>
</tr>
<tr>
<td></td>
<td>• Full DSP/BIOS aware debugging</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BIOS Visualization Tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Kernel Object Viewer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BIOS Configuration Tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Supports ARM+DSP and DSP-only devices</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Optimizing TI Optimizing DSP Compiler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Wizards to help with building and verifying xDAISxDM algorithms (i.e. codecs)</td>
<td></td>
</tr>
</tbody>
</table>

**Bottom Line:**

- Go with GHS for an all-in-one ARM+DSP development tool
- Choose TI for “Best-in-Class” DSP development at a lower price point
Linux Distributions

What are distributions?

Build It Yourself?

Quote from kernel.org:

If you're new to Linux, you don't want to download the kernel, which is just a component in a working Linux system. Instead, you want what is called a distribution of Linux, which is a complete Linux system.

There are numerous distributions available for download on the Internet as well as for purchase from various vendors; some are general-purpose, and some are optimized for specific uses.

- This may be a bit of an understatement – even experienced users usually use a distribution
- Creating a distribution takes a lot of effort
- Maintaining a distribution ... takes even more effort
- In fact, using a distribution even takes quite a bit of effort
What Is a ‘Linux Distribution’

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

1. **Linux kernel** port
   - A TI LSP or Linux PSP is a Linux kernel port to a device, not just a set of device drivers

2. **Bootloader**
   - Uboot is the standard bootloader for ARM Linux

3. **Linux ‘file system’**
   - This does NOT mean a specific type of file system like FAT file system or flash file system … rather, it more like the “C:" drive in Windows
   - It refers to all the ‘user mode’ software that an application needs such as graphics libraries, network applications, C run-time library (glibc, uclibc), codec engine, dynamically-loaded kernel modules (CMEM, DSPLINK)

4. **Development tools**
   - GCC, GDB
   - MV DevRocket, CCS, GHS Multi, etc.

Linux isn’t complete without a distribution

- **MontaVista** and **Timesys**, for example, provide commercial (i.e. production) distribution for TI’s DaVinci / OMAP processors

- A few distributions supporting the open-source BeagleBoard (OMAP35x-based) include: **OpenEmbedded**, Ubuntu, Fedora, Android, Gentoo, ARMed slack and ALIP
O/S Choices

Linux Distributions Options for TI

<table>
<thead>
<tr>
<th>Commercial</th>
<th>Community</th>
<th>Custom (Build it Yourself)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Timesys</td>
<td>TI SDK (PSP)</td>
<td>Open Embedded (OE)</td>
</tr>
<tr>
<td>• MontaVista</td>
<td>Ångström</td>
<td>Custom from Sources</td>
</tr>
<tr>
<td>• Etc.</td>
<td>• Binary</td>
<td>• &quot;GIT&quot; from kernel.org, and others</td>
</tr>
<tr>
<td></td>
<td>(Update patches)</td>
<td>• Bit-Bake</td>
</tr>
</tbody>
</table>

Ease of Use

• Easy
• Tested

Experienced User
• Latest
Community Options

**Community Options**

- **TI Linux SDK (PSP)**
  - Pre-built snapshot of Linux tested against specific version of TI Software Development Kits
  - Updated at each new SDK/DVSDK release
  - PSP = Platform Support Package (name of TI team)
  - Currently, a “BusyBox-based” bare-bones distro (“lean/mean”)
  - Arago open-source OE project
    - Advantage of OE – recipies can be reused by Angstrom (or custom OE) users
    - In general, users shouldn’t (re)build using OE; no reason to, because if you want more features, we recommend you go with Angstrom (also built using OE)

- **Ångström**
  - Open-source, full-featured Linux distro targeted for embedded systems
  - Get it from:
    - User-compiled binaries widely available for many targets
    - [Narcissus](http://www.angstrom-distribution.org/narcissus)
  - Web-based tool creates binary vers (w/ your own package sel’n)
  - Built using OE (user community can re-use TI OE recipies)

---

**Ångström : Narcissus**

![Narcissus Image](http://www.angstrom-distribution.org/narcissus)
DIY Options

- **Open-Embedded (OE)**
  - Build Linux from source using OE’s Bit-Bake recipe(s)
  - Many recipes available for various architectures, including many variations for a given device
  - Builds full-up distro including Kernel and Filesystem
  - TI builds it’s PSP Linux distro’s via OE

- **Build from Sources** *(roll your own)*
  - Build directly from sources, such as kernel.org
  - Use GIT, as well as SVN and others to get sources from repo’s
  - *Are you nuts? Either you want to waste your life redoing what OE did, or you’re so advanced you don’t need this presentation.*
Side-bar: Open-Embedded

DIY Linux: Open Embedded (OE)

- u-boot
- Linux application packages on internet
- OpenEmbedded Recipes, config files
- Code Sourcery
- Commercial Linux OS and tools vendors
- Code Sourcery

TI’s New Linux Strategy

- u-boot
- Linux application packages on internet
- TI Linux SDK
- TI Customers
- Code Sourcery
- Commercial Linux OS and tools vendors
- Code Sourcery

TI Device Linux Kernel Patching Methodology

Arago “Staging” Area
http://arago-project.org/git/projects
- TI ‘work’ area
- OMAP3x Linux patches
- OMAP-L1x Linux patches
- DaVinci Linux patches
- U-Boot patches
- Test scripts and framework
- Earliest customer access to patches

DaVinci and OMAP Linux staging trees
http://tgip.kernel.org/?p=linux-kernel/git/davinci
- Current Linux kernels plus recently accepted TI and third-party patches
- Small temporary deltas from mainline kernel

Mainline Trees
- ‘Official’ Linux kernels rel’s
- [mach-omap2]
- [mach-davinci]
- ‘Official’ U-Boot releases

Accepted Patches

You
Commercial Options

Commercial O/S Vendors

- **Linux**
  - MontaVista
  - TimeSys
  - Wind River
  - Mentor
  - Ridgerun

- **WinCE**
  - Adeneo
  - Mistral
  - MPC Data
  - BSQUARE

- **RTOS**
  - Green Hills
  - Wind River (VxWorks)
  - ELogic (ThreadX)
  - QNX
  - Mentor (Nucleus)

Linux Partner Strategy

- **Commercial**: provide support, off-the-shelf Linux distributions or GNU tools
- **Consultants**: provide training, general embedded Linux development expertise, or specific expertise for developing drivers or particular embedded applications
- [http://www.tiexpressdsp.com/index.php/Linux_Consultants_and_Commercial_Linux_Providers](http://www.tiexpressdsp.com/index.php/Linux_Consultants_and_Commercial_Linux_Providers)

Commercial vs Community

- **Commercial**
  - Less effort – another team does the work of porting new kernel to the distribution... and then laboriously testing it over-and-over again
  - More robust – many customers generating more inputs/errata to team testing an maintaining the distribution
  - More secure – among other reasons, many homebrew distributions don’t get around to adding security patches due to effort and time
  - Latest features? Many vendors *backport* new features into their kernels – thus, you get the stablility of a known kernel but with new features
  - Good choice if: you don’t need the absolute latest features; you have a many projects to amortize the costs; you’re a Linux wiz who really knows what they’re doing.
  - **Bottom Line** – Commercial distributions trade cost (and the bleeding edge features) for robustness and less-effort. What is it worth, if your company depends on this product?

- **Community** *(to Git or not)*
  - Access to latest improvements in Linux kernel
  - Want to know exactly how it is all put together
  - Maximum involvement in Linux community
  - No costs ... (unless you count your labor)
  - **Bottom Line** – Choose this option if you have the Linux expertise and labor is cheaper than NRE; or, you need access to the very latest features
Example Comparison: **MVL Pro 5.0 vs GIT**

<table>
<thead>
<tr>
<th></th>
<th>MVL 5.0 Pro</th>
<th>Community Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel Version</strong></td>
<td>Uses 2.6.18, which is almost 3 years old</td>
<td>Uses latest available kernel</td>
</tr>
<tr>
<td><strong>Kernel bug-fixes</strong></td>
<td>Applied to 2.6.18, so no need to change kernel versions</td>
<td>Applied to current release, which changes every few months. User may need new kernel to get a fix.</td>
</tr>
<tr>
<td><strong>File System</strong></td>
<td>Comprehensive host and target file systems with GUI tools for optimization.</td>
<td>Not part of kernel. TI is addressing through Arago. Initially may be less user-friendly than MVL.</td>
</tr>
<tr>
<td><strong>Linux run-time Licensing</strong></td>
<td>Demo copy and LSP open source, but original licensing has created confusion.</td>
<td>TI offering is clearly free as GIT Linux distributions are open source.</td>
</tr>
<tr>
<td><strong>Tools licensing</strong></td>
<td>GNU Tools free. IDE requires annual subscription.</td>
<td>GNU Tools free. IDE requires annual subscription.</td>
</tr>
<tr>
<td><strong>Third-party support</strong></td>
<td>MV and its partners</td>
<td>Multi-vendor, including MV</td>
</tr>
</tbody>
</table>

**TI Customers Can CHOOSE a Linux Path:**

**Community or Commercial**

- **Community first path**
  - TI delivers LSP/DVSDK to community
  - Smaller set of applications
  - Customer builds up solution
  - Open source assets
  - Customer assets
  - Faster access, newer kernels
  - More customer responsibility
    - Invest own resources vs. $$

- **Commercial complement path**
  - Commercial Linux partner pulls from community
  - Partner adds value: production testing, tools integration, support, application bundles, etc. for customers
  - Service and subscription sales
  - Executing with MontaVista, Timesys...
  - Opportunities for other commercial vendors
**TI SDK 4.0 (PSP Linux : Root Filesystem)**

- Integrated GUI and multimedia stacks based on industry standard components:
  - GUI and graphics: QT
  - Multimedia: gstreamer
- Many common Linux utilities and programs either in busybox or provided separately
- Additionally, the goal is to simplify the installation of the SDK software

---

**Other Linux packages**
Alsa, binutils, curl, dosfstools, e2fsprogs, ..., gdbserver, ncurses, opkg, oprofile, scp, ssh, ...

**Busybox**
See: [http://www.busybox.net/screenshot.html](http://www.busybox.net/screenshot.html)

**Linux Kernel**

---

![Texas Instruments logo](http://www.busybox.net/)
(Optional) How Do I Rebuild Linux … and Why?

How Do I Build It, Let Me Count the Ways...

1. Don’t … find a pre-built Linux uImage
2. Build Default Linux
   a. make defconfig
   b. make uImage

Why Re-Build Linux Kernel?

- TI SDK’s often support various ARM CPU’s, thus GSG directs you to specify target processor and rebuild kernel
- You made changes to a Linux source file (i.e. driver, etc.)

Change to Kernel’s Directory  (TI/MontaVista LSP Example)
> cd ti-davinci/linux-2.6.18_pro500

Configure the Kernel
> make ARCH=arm CROSS_COMPILE=arm_v5t_le- davinci_dm644x_defconfig

Build the Kernel
> make ARCH=arm CROSS_COMPILE=arm_v5t_le- uImage
How Do I Rebuild Linux … and Why?

Configure the Kernel

```bash
host $ cd ti-davinci/linux-2.6.18_pro500
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le- davinci_dm644x_defconfig
```

Verify Kernel

```bash
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le- checksetconfig
```

Customize the Kernel

```bash
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le- menuconfig
```

Build the Kernel

```bash
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le- uImage
```

Build Loadable Modules (i.e. dynamic “insmod” modules)

```bash
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le- modules
```

```bash
host $ make ARCH=arm CROSS_COMPILE=arm_v5t_le
    INSTALL_MOD_PATH=/home/<useracct>/workdir/filesys
    modules_install
```

How Do I Build It, Let Me Count the Ways…

1. **Don’t** … find a pre-built Linux uImage
2. **Build Default Linux**
   a. make defconfig
   b. make uImage
3. **Build ‘Custom’ Linux**
   a. make defconfig
   b. make menuconfig
   c. make uImage
Building Programs with gMake

Introduction

DaVinci software can be built using GNU’s standard gMake utility. In the future, when we introduce building codec engine servers, we’ll invoke another tool – XDC to build the application. In one part of the lab, you will be introduced to Configuro, and XDC tool that allows you to consume packaged content from TI (or 3rd parties). It will create some files used by gMake to build the final application.

This chapter introduces the GNU gMake utility and how to build a makefile from scratch. You will then have a chance to analyze the makefile that will be used throughout the rest of the workshop.

Learning Objectives

Outline

- Brief overview of gcc for compiling and linking
- Understand how to build targets using gmake
- Use rules and variables (built-in, user-defined) in makefiles
- Learn how to add “convenience” and “debug” rules
- Learn how to handle C (header file) dependencies
- Use Configuro to consume packaged content
Chapter Topics

Building Programs with gMake ................................................................. 5-1

Building Programs With gMake ............................................................... 5-3
Big Picture .............................................................................................. 5-3
Creating/Using a Makefile ................................................................. 5-4
Using Variables and Printing Debug Info ........................................... 5-6
Wildcards and Pattern Substitution ..................................................... 5-9
Basic Makefile – Code Review .............................................................. 5-10
Handling Header File Dependencies ................................................... 5-11

Using Configuro to Consume Packages .............................................. 5-12
Current Flow ......................................................................................... 5-12
What is a Package? ............................................................................. 5-13
Configuro – How it Works ................................................................. 5-13
Using Configuro in the Upcoming Lab ............................................... 5-14
MakeFile Example – Using Configuro ............................................... 5-15
Looking Inside Compiler.opt & Linker.cmd ...................................... 5-16

Lab 5 – Using gMake ............................................................................ 5-17
Lab05ab_basic_make ........................................................................... 5-18
Lab05c_x86_configuro .......................................................................... 5-27
Lab05d_standard_make ........................................................................ 5-31
Lab05e_optional_challenge .................................................................. 5-34

Appendix ................................................................................................. 5-49
Building Programs With gMake

Big Picture

**Build Overview**

- Source Files
  - app.c
  - app.cfg
  - app.h

- Build Instructions
  - Build Tool

- Executables
  - Targets
    - app.x64P
    - app.x470MV
    - app.x86U

**Build Tool Goals:**
1. Build executable (target) from input files (dependencies) using build instructions (commands)
2. Build for multiple targets at once (e.g. ARM, X86, DSP)

**Solution:** command line (e.g. cl6x, gcc) or scripting tool (gMake, etc.)

**Command Line (Examples 1-2)**

- Dependencies
  - app.c
  - app.o

- Commands
  - Command Line

- Targets
  - app.o
  - app.x86U

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

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

- Might be more convenient to place commands in a script/batch file...makefile...
Creating/Using a Makefile

Basic Makefile with Rules

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

\[
\text{TARGET : DEPENDENCY TAB COMMANDS...}
\]

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

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

- Becomes:

\[
\text{RULE}
\]

Creating Your First Makefile

- Command Lines

```bash
gcc -c -g app.c -o app.o
gcc -g app.o -o app.x86U
```

- Makefile

```bash
# Makefile for app.x86U (goal)
app.x86U : app.o
  gcc -g app.o -o app.x86U
app.o : app.c
  gcc -c -g app.c -o app.o
```
Running gMake

“Dependencies”  
appl.c  
appl.o  

“Target”  
appl.o  
appl.x86U  

To run gMake, you can use the following commands:

- `make` (assumes the makefile name is “makefile”, runs FIRST rule only)
- `make appl.x86U` (specifies name of “rule” – e.g. appl.x86U)
- `make -f my_makefile` (can use custom name for makefile via forcing flag... -f)

- gMake looks at timestamps for each target and dependency. If the target is newer than its dependencies, the rule (and associated commands) will not be executed.

- To “rebuild all”, use the “clean” rule to remove intermediate/executable files...

“Convenience” Rules

“Build All Targets”

```
.PHONY : all
all : appl.x86U ...(all “goals” listed here)
```

“Remove Unwanted Files”

```
.PHONY : clean
clean :
    rm -rf appl.o  
    rm -rf appl.x86U
```

“Copy Executable to the install directory”

```
.PHONY : install
install : appl.x86U
    cp appl.x86U /dir1/install_dir
```

Note: “all” rule is usually the first rule because if you type “make”, only the first rule is executed.

Convenience rules (e.g. all, clean, install) can be added to your makefile to make building/debug easier.

For example, a “clean” rule can delete/remove existing intermediate and executable files prior to running gMake again.

If the rule’s target is NOT a file, use the _PHONY directive to tell gMake not to search for that target filename (it’s a phony target).

Let’s look at three common convenience rules (to use, type “make clean”):

- Remove Unwanted Files
- Copy Executable to the install directory
- Build All Targets

DaVinci / OMAP Workshop - Building Programs with gMake
gMake Rules Summary

- 3 common uses of rules include:
  - [.x] – final executable
  - [.o] – intermediate/supporting rules
  - [.PHONY] – convenience rules such as clean, all, install

Examples:

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

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

- `.PHONY`
  ```
  clean :
  rm -rf app.x86U
  ```

Run:
- `make` (assumes makefile name is “makefile” or “Makefile” and runs the first rule only)
- `make app.x86U` (runs the rule for app.x86U and all supporting rules)
- `make clean`

Using Variables and Printing Debug Info

Using Built-in Variables

- **Simplify your makefile** by using these built-in gMake variables:
  - `$@` = Target
  - `$^` = All Dependencies
  - `$<` = 1st Dependency Only

- Scope of variables used is the current rule only.

Example:

Original makefile...

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

Becomes...

```
app.x86U: app.o
 gcc -g $^ -o $@
```
User-Defined Variables & Include Files

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

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

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

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

Printing Debug/Warning Info

- Two common commands for printing info to stdout window:
  - `echo` – command line only, flexible printing options ("@" suppresses echo of "echo")
  - `$(warning)` – can be placed anywhere in makefile – provides filename, line number, and message
- **Examples:**

  ```makefile
  app.x86U : app.o
          $(CC) $(CFLAGS) $(LINK_FLAGS) $@
          @echo
          @echo $@ built successfully; echo
  $(warning Source Files: $(C_SRCS))
  app.x86U : app.o $(warning now evaluating dep’s)
          $(CC) $(CFLAGS) $(LINK_FLAGS) $@
          $(warning $@ built successfully)
  ```

- `$(*warning)` does not interrupt gMake execution
- A similar function: "$(error)" stops gMake and prints the error message.
Quiz

- Fill in the blanks below assuming (start with .o rule first):
  - Final "goal" is to build: main.x86U
  - Source files are: main.c, main.h
  - Variables are: CC (for gcc), CFLAGS (for compiler flags)

```
CC := gcc
CFLAGS := -g

# .x rule
_________ : _______  ________  ___ -o ___
         :  _____  ________  ___ -o ___

# .o rule
_________ : _______  ________
         :  _____  ________ -c ___ -o ___
```
Wildcards and Pattern Substitution

Using “Wildcards”

- **Wildcards** (*) can be used in the *command* of a rule. For example:

  ```
  clean :
  rm -rf *.o
  ```

  Removes all .o files in the current directory.

- Wildcards (*) can also be used in a *dependency*. For example:

  ```
  print : *.c
  lpr -p $?
  ```

  Prints all .c files that have changed since the last print.

  Note: automatic var "$?" used to print only changed files

- However, wildcards (*) can **NOT** be used in *variable declarations*. For example:

  ```
  OBJS := *.o
  ```

  OBJS = the string value "*.o" – not what you wanted

  To set a *variable* equal to a list of object files, use the following *wildcard function*:

  ```
  OBJS := $(wildcard *.o)
  ```

Simplify Your MakeFile Using “%”

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

  **Original Makefile**
  ```
  app.x86U : app.o main.o
  $(CC) $(CFLAGS) $^ -o $@
  
  app.o : app.c
  $(CC) $(CFLAGS) -c $^ -o $@
  
  main.o : main.c
  $(CC) $(CFLAGS) -c $^ -o $@
  ```

  **Makefile Using Pattern Matching**
  ```
  app.x86U : app.o main.o
  $(CC) $(CFLAGS) $^ -o $@
  
  %.o : %.c
  $(CC) $(CFLAGS) -c $^ -o $@
  ```

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

**Basic gMake Makefile – Review (1)**

- Include file that contains tool paths (e.g., the path to gcc)
- User-defined variables
  - CC := $(X86_GCC_DIR)gcc
  - CFLAGS := -g
  - LINKER_FLAGS := -lstdc++
- "all" rule
- Main "goal" of makefile...
- Intermediate .o rule. Notice the use of pattern matching.

**Basic gMake Makefile – Review (2)**

- "clean" rule. Removes all files created by this makefile. Note the use of .PHONY.
- "printvars" rule used for debug. In this case, it echoes the value of variables such as "CC", "CFLAGS", etc.
Handling Header File Dependencies

Handling Header File Dependencies (1)

- Handling header files dependencies can be difficult – especially if you have to edit the header filenames manually.
- If a header file gets updated, but is not specified in the makefile, the change will not trigger a new .o target to be rebuilt.
- Let's see how to handle this properly...

```
app.c
#include "app.h"
...
```

```
makefile
app.x86U : app.o
  $(CC) $(CFLAGS) $^ -o $@
app.o : app.c app.h
  $(CC) $(CFLAGS) -c <$ < -o $@
```

- Which dependency is missing from the makefile above?
- Do you want to manually specify all the header files for all .c programs in your application? Or would something automatic be of interest? ...

Handling Header File Dependencies (2)

- Which common build tool is really good at locating header files in .c files and building a list? gcc, of course.
- So, let's use the compiler to create a list of header file dependencies and place these in a corresponding .d file:

```
%.d : %.c
  $(CC) -MM $(CFLAGS) < > @
  @sed -e 's|.*:|$*.o:|' < $@ >> $@

-in app.d
```

- -MM option creates a list of header files from a .c file and echos them to the stdout window – which we pipe into a corresponding .d file (using > $@).
- The cryptic “sed” command is used to reformat the list of files in the .d file (e.g. app.h) to look like a rule for app.o:

```
app.d
app.o : app.h
```

- Three notes about the -include:
  - Includes the new app.d file which contains the proper header file dependencies.
  - The “-” prevents us from getting an error on first pass before .d file is created.
  - Remember, any time an included file is regenerated, make processing starts over.
### Code for Handling Dependencies

```
C_SRCS := $(wildcard *.c) # Create list of .c files
OBJS := $(subst .c,o,$(C_SRCS)) # Create list of .o files
DEFS := $(subst .c,d,$(C_SRCS)) # Create list of .d files from .c files

# 3. Dependency Rule (.d)
# -----------------------
%.d : %.c
@echo "Generating dependency file"
$(CC) -MM $(CFLAGS) $<  > $@
# Use gcc to generate list of dep's
@echo "Formatting dependency file"
$(call format_d, $@, $(PROFILE))
# Call macro to format dep file list

# Define Macro which formats .d file as gmake rule
define format_d
@mv -f $1 $1.tmp
@sed -e 's|.*:|$2$*.o:|' < $1.tmp > $1
@rm -f $1.tmp
endef

# Include dependency files
# Only include (.d) files if "clean" is target -- avoids warning
ifneq ($(filter clean,$(MAKECMDGOALS)),clean)
-include $(DEFS)
endif
```

---

### Using Configuro to Consume Packages

#### Current Flow

- **Our current build flow looks something like this:**
  ```
  app.c
  #include <stdio.h>
  ...
  Compiler (gcc)
  Linker (gcc)
  app.x86U
  ```

- **How does gcc know where to find the following?**
  - Header files: e.g. stdio.h
  - Libraries: e.g. stdio.lib

- **gcc needs path information for both files:**
  ```
  app.c
  #include <stdio.h>
  ...
  Compiler (gcc)
  Linker (gcc)
  app.x86U
  ```

- **So, the compiler needs to know the paths to the header files (e.g. -i ...) and the linker needs to know the paths to the libraries (e.g. -l ...).**
What is a Package?

**Goal: Consume “Packages” from TI & 3rd Parties**
- TI (and 3rd party) content is delivered as a "package".
- Package = library + metadata = "smart" library

- **Packages contain:**
  - Payload (i.e. library)
  - Program metadata
    - Version Info
    - Dependencies
    - Library/header references
    - Module (interface description)

- gcc STILL needs **path** information from this "package":

- How does gcc know where to get this path information from?

**Configuro – How it Works**

**Using Configuro**
- **Configuro** is a tool that helps users consume/use packaged content.

- **Configuro creates** two files:
  - compiler.opt – includes compiler options and header file paths
  - linker.cmd – includes linker options and library file paths

- **Configuro needs** four other inputs to help it perform properly:
  - .cfg – indicates which packages to include
  - XDCPATH – list of paths where all the packages are located
  - Platform – e.g. ti.platforms.evmDM6446
  - Target – e.g. gnu.targets.MVArm9
Using Configuro in the Upcoming Lab

Using Configuro

- We’re using Configuro in the upcoming lab for two reasons:
  1. To use a more efficient printf() – System_printf(). This will help us understand the basics of using Configuro.
  2. TI and our 3rd parties deliver content as “packages” – so, you’ll need to know the proper use of Configuro to consume them (many more examples come later in the workshop).

- Shown below are: app.c and app_cfg.cfg
  - Notice the inclusion of “System.h” – which comes from a “package” delivered by TI. This library allows the use of System_printf():

```c
#include <stdio.h>
#include <xdc/runtime/System.h>
#include "app.h"

int main(void)
{
    System_printf("Hello World (xdc)\n");
    printf("Hello World %d (std)\n", YYYY);
    return 0;
}
```

/* app_cfg.cfg */
var mySys = xdc.useModule('xdc.runtime.System');

```text
app_c and app_cfg.cfg
```

```
app.c
/* app_cfg.cfg */
var mySys = xdc.useModule('xdc.runtime.System');
```
MakeFile Example – Using Configuro

```bash
#--------------------------
#-- Configuro variables --
#--------------------------
XDCROOT := $(XDC_INSTALL_DIR)
CONFIGURO := $(XDCROOT)/xs xdc.tools.configuro
export XDCPATH := /home/user/rtsc_primer/examples;$(XDCROOT)
CONFIG := app_cfg
TARGET = gnu.targets.Linux86
PLATFORM = ti.platforms.PC

#--------------------------
#-- .cfg rule            --
#--------------------------
%/linker.cmd %/compiler.opt : %.cfg
$(CONFIGURO)   –c   $(CC_ROOT)   –t   $(TARGET)   –p   $(PLATFORM)   -o   $(CONFIG)   $<

#--------------------------
#-- .x and .o rules      --
#--------------------------
app.x86U : app.o $(CONFIG)/linker.cmd
 $(CC) $(CFLAGS) $^ -o $@

%.o : %.c $(CONFIG)/compiler.opt
 $(CC) $(CFLAGS) $(shell cat $(CONFIG)/compiler.opt) –c $< -o $@

#--------------------------
#-- .cfg rule            --
#--------------------------
%/linker.cmd %/compiler.opt : %.cfg
$(CONFIGURO)   –c   $(CC_ROOT)   -t   $(TARGET)   -p   $(PLATFORM)   -o   $(CONFIG)   $<
```

How are the created files (linker.cmd, compiler.opt) used in the makefile? ...
Looking Inside `Compiler.opt` & `Linker.cmd`

### Compiler.opt

- `-I"/home/user/rtsc_primer/examples"
- `-I"/home/user/dvsdk_1_30_00_40/xdc_3_05"
- `-I"/home/user/dvsdk_1_30_00_40/xdc_3_05/packages"
- `-I"/home/user/dvsdk_1_30_00_40/xdc_3_05/packages"
- `-I"..
- `-Dxdc_target_types__="gnu/targets/std.h"
- `-Dxdc_target_name__=MVArm9
- `-Dxdc_cfg__header__="/home/user/lab05d_standard_make/app/debug/mycfg/package/cfg/mycfg_x470MV.h"

### Linker.cmd

```bash
INPUT(
    /home/user/lab05d_standard_make/app/debug/mycfg/package/cfg/mycfg_x470MV.o470MV
    /home/user/rtsc_primer/examples/acme/utils/lib/acme.utils.a470MV
    /home/user/dvsdk_1_30_00_40/xdc_3_05/packages/gnu/targets/rts470MV/lib/gnu.targets.rts470MV.a470MV
) ```
Lab 5 – Using gMake

Goal

Welcome to the compulsory “Hello World!” lab, where we will begin our exploration of the DaVinci Evaluation Module (DVEVM) and the software programming tools available to you in your development. 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-926-based DaVinci target system using Linux terminals.

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
Lab 5 – Using gMake

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.

Exercise

The exercise that was planned for this section was replaced by the quiz in the material. If you are new to GNU make, you might want to refer back to the quiz as you begin Part A of the lab…

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 workshop directory and list its contents.

Descend into the /home/user/workshop directory using the “cd” command. (“cd” is short for “change directory”).

Use the “ls” command to survey the contents of this directory (lower case “LS” is short for “list”):

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

   cd ~/workshop/lab05ab_basic_make
   - or -
   cd /home/user/workshop/lab05ab_basic_make
   - or, since you’re probably already in the workshop 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 (`app.x86U`) indicates we are building for the x86 (or host PC). In the future, we will build for the ARM target (MontaVista Linux) on the DaVinci DVEVM and it will have a different extension (`app.x470MV`).

   **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 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:

```plaintext
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:

```plaintext
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:

```plaintext
# ---------------------------------------------------
# ------ 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:

```plaintext
make
```

After running make, list the current directory.

```plaintext
ls
```

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

```plaintext
./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 app.x86U
    rm -rf app.o
```

`.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).

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 DaVinci 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":

```makefile
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:

```makefile
app.o : app.c
 $(CC) $(CFLAGS) -c app.c -o app.o
```

- to -

```makefile
%.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:

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

- to -

```makefile
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 /workshop/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 in to stdout when we run the application.
36. **Open setpaths.mak and browse the contents.**

Migrate up two levels to the workshop 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 name of the path to the Linux 86 gcc compiler? __________________________
- What is the 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:

```makefile
–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:

```makefile
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 DVEVM
39. **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 “*Configuro Vars*”.

**Hint:** “Configuro Vars” was the section we just asked you to copy in the previous step.

40. **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:

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

In the .o rule, add the following dependency:

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

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

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

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

41. **A little quiz 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 $^ to indicate both dependencies. If you use $<, *gMake* will not produce the proper output. Explain:

_________________________________________________________________
_________________________________________________________________
42. 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 “clean” rule to remove Configuro’s intermediate files.

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

rm –rf $(CONFIG)

43. 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.

If you’re satisfied with the results, time to move on to Part D – the last part of this lab…

44. What other functions are in the system package?

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

~/dvsdk_2_00_00_22/xdctools_3_10_03/packages/xdc/runtime/System.h

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

45. Introducing the parent and child 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 ARM9 target on the DVEVM. An install rule exists that automatically copies the executables to the proper directory on the DVEVM 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, configure, .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.

46. 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 ~/workshop/lab05d_standard_make/app

47. 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.x470MV? 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.)
48. Perform a “make clean” and observe the messages on the screen.

49. Using “help”.

The authors built in some “help” information. Try:

```make
make help
```

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

50. Make “all”.

Type:

```make
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.x470MV` and `app_RELEASE.x470MV`. You can’t run these on an x86 PC, but next we will install them to the DVEVM so that we can run them to test if they are working properly.

51. Make “install”.

Run a “make clean” first, then try:

```make
make install
```

Executing just the install rule will automatically create both versions of the application and install them to `/opt/workshop` directory on the DVEVM (which is `/home/user/workdir/filesys/opt/workshop` within Ubuntu Linux). If you don’t have a terminal open, open a terminal to the DVEVM using Tera Term. Log in as “root” and change to the `/opt/workshop` directory. Do you see the two executables?

Run both of them and see if they work. You will need a “./” in front of the filename for Monta Vista Linux to recognize the filenames.

```bash
ll (lowercase LL – is an alias for ls -lisa)

./app_DEBUG.x470MV

./app_RELEASE.x470MV
```
52. **Let’s turn on some debug stuff…**

   The parent makefile allows you to specify debugging commands on the command line. Let’s try the 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
   ```

53. **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)?

   * ________________________________________________________________
Lab5e_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.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.

54. 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.
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.
55. **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 *workshop* 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 &
```

56. **“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.

57. **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.
Step 56
Need to include our tool paths. They are defined in setpaths.mak and located such that this file is common for all labs.

Step 57
Along with the build flags for ARM9 gcc, we see there are two sets of flags associated with our "profile" choices (debug, and release).
58. 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.

59. 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.

60. 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.

61. 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”.

62. 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.
Lab 5 – Using gMake

makefile_profile.mak (cont’d 2)

# 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
#
# 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

DaVinci / OMAP Workshop - Building Programs with gMake
63. 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 stdout.

64. 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.


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)

```makefile
# ****************************************************************************
# 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) $(CC) $(CFLAGS) $(LINKER_FLAGS) $^ -o $@
@echo "Successfully created: $@ 
```

---

**Step 63**
Default_Rule is the default rule ... wait do I hear an `@echo` here?

**Step 64**
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 65**
Similar to the .x rule, we've added profile/path vars to the .o rule
66. 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.

67. 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).
Lab 5 – Using gMake

DaVinci / OMAP Workshop - Building Programs with gMake

5 - 43

makefile_profile.mak (cont’d 4)

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 script's feedback, the Configuro's
  feedback is piped into 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) $< \n> $(PROFILE)/$(CONFIG)_results.log
endif
@echo "Configuro has completed; it's results are in $(CONFIG) " ; echo
```

The "no" .cfg rule

```
%.cfg :
$(AT) touch $(CONFIG).cfg
```
68. 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)**

```makefile
# "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."

# "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
```

**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.
69. **Macro: format_d**

   As stated before, this macro reformatms 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.

70. **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)

Macros

# ****************************************************************************
#
#    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

---

Step 69
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 70
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"
makefile_profile.mak (cont’d 7)

```make
# ****************************************************************************
#    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
```

71. **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 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.)

*Wilbur* you need it or not, GNU Make (i.e. gMake) is the most popular scripting language out there. Later in the workshop, we’ll see an alternative scripting tool from TI, called XDC.
Appendix

Here are the answers to the quiz from the chapter material:

Quiz

◆ Fill in the blanks below assuming (start with .o rule first):
  - Final “goal” is to build: main.x86U
  - Source files are: main.c, main.h
  - Variables are: CC (for gcc), CFLAGS (for compiler flags)

```makefile
CC := gcc
CFLAGS := -g

# .x rule
main.x86U : main.o
  $(CC) $(CFLAGS) $^ $@

# .o rule
main.o : main.c main.h
  $(CC) $(CFLAGS) -c $< -o $@
```

◆ Could $< be used in the .x rule? What about $^ in the .o rule?

Yes, the $< can be used in the .x rule because there is only ONE dependency. However, the .o rule has two dependencies and would therefore need $^.

Answers to the lab quiz questions:

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:

The .o rule is running the COMPILER. It only knows how to compile a .c file. Compiler.opt is NOT a .c file, so we must use $< to indicate just the first dependency. If $^ is used, the compiler will attempt to compile the compiler.opt file and crash. However, we need compiler.opt contents to show up on the command line, hence the $(shell cat …).

Now look at the .x rule. Study the command that contains $(CC). Notice that this time we use $^ to indicate both dependencies. If you use $<, gMake will not produce the proper output. Explain:

Both dependencies are needed. The linker.cmd file is an INPUT to the linker and therefore is required. So, $^ must be used and $< (first dependency only) would fail.
We hate wasting paper, too. Even so, blank pages like this provide necessary padding to keep the odd page out.
Intro to Device Drivers

Introduction

By the end of this chapter, we should have an audio driver setup to capture and/or playback audio data. Along the way, we will also learn about the basic concepts of Linux processes and device drivers.

Outline

Learning Objectives

- Differential between Kernel and User Space
- Understand two methods for adding modules to the Linux kernel
- Define nodes in Linux and why they are useful
- Describe why a filesystem is needed, and which one is used for the DVEVM HDD
- Describe the basic file and driver I/O interfaces
- List two dev’s that constitute the OSS driver
- Build an audio pass-thru application – given audio input and output examples
Chapter Topics

Intro to Device Drivers

Kernel vs User Space

Linux Drivers - Basic Concepts

(1) Load the driver code into the kernel
(2) Create a virtual file reference (node)
(3) Mount block drivers using a filesystem
(4) Access resources using open/close/read/write

Linux OSS (audio) Driver

Lab 6 - Using the OSS Driver

Lab06a_audio_record
Lab06b_audio_playback
Lab06c_audio_loopthru
(Optional) Lab06d_challenge1

Driver Documentation
Kernel vs User Space

Kernel Space
- The Linux kernel manages the machine's hardware
- The kernel, in particular its device drivers, form an interface between the end-user/programmer and the hardware
- Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of kernel space

User Space
- End-user programs, like the BASH shell, are part of the user space.
- User applications cannot interact directly with the system's hardware, but do so through the kernel supported functions
Four steps are required for users to access kernel space drivers:

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

### Kernel Object Modules

How to add modules to Linux Kernel:

1. Static (built-in)

   ```
   Linux Kernel
   oss fbdev httpd
   v4l2 nfsd dsp ext3
   ```

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

2. Dynamic (insmod)

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

   - Use `insmod` (short for insert module) command to dynamically add modules into the kernel
   - Keep statically built kernel small (to reduce size or boot-up time), then add functionality later with insmod
   - Insmod is also handy when developing kernel modules
   - Later we'll insert two modules (cmem.ko, dsplink.ko) using a script: `loadmodules.sh`

### Static Linux Kernel Configuration

```
#> make ARCH=arm CROSS_COMPILE=arm_v5t_le- menuconfig
```
(2) Create a virtual file reference (node)

### Linux Driver Registration

<table>
<thead>
<tr>
<th>Command: mknod</th>
<th>Description: Make Node command.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax:</strong></td>
<td><code># mknod &lt;name&gt; &lt;type&gt; &lt;major&gt; &lt;minor&gt;</code></td>
</tr>
<tr>
<td><strong>&lt;name&gt;:</strong></td>
<td>Node name (i.e. virtual file name)</td>
</tr>
<tr>
<td><strong>&lt;type&gt;:</strong></td>
<td>block — character</td>
</tr>
<tr>
<td><strong>&lt;major&gt;:</strong></td>
<td>Major number for the driver</td>
</tr>
<tr>
<td><strong>&lt;minor&gt;:</strong></td>
<td>Minor number for the driver</td>
</tr>
</tbody>
</table>

**Example:**

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

**Usage:**

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

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

#### Block and Character Drivers

<table>
<thead>
<tr>
<th>Block Drivers:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/hda</td>
<td>ATA → harddrive, CF</td>
</tr>
<tr>
<td>/dev/ram</td>
<td>external RAM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character Drivers:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/dsp</td>
<td>sound driver</td>
</tr>
<tr>
<td>/dev/video0</td>
<td>v4l2 video driver</td>
</tr>
<tr>
<td>/dev/fb/0</td>
<td>frame buffer video driver</td>
</tr>
</tbody>
</table>

- **Block drivers allow out of order access**
- **Block devices can be mounted into the filesystem**
- **Character drivers are read as streams in a FIFO order**
- **Networking drivers are special drivers**
Linux Device Registration

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

```
/ # cd /dev
/dev # ls -lsa
 0 brw-rw---- 1 root disk  0,  0 Jun 24  2004 /dev/hda
 0 crw-rw---- 1 root uucp  4, 64 Mar  8  2004 /dev/ttyS0
 0 crw------- 1 user root 14,  3 Jun 24  2004 /dev/dsp
 0 crw------- 1 user root 29,  0 Jun 24  2004 /dev/fb/0
 0 crw------- 1 user root 29,  1 Jun 24  2004 /dev/fb/1
```

Permissions (user, group, all) Minor number Name
Major number /dev directory

(3) Mount block drivers using a filesystem

Mounting Block Devices

```
/ # mkdir /mnt/harddrive
/ # ls /mnt/harddrive
/ # mount -t ext3 /dev/hda1 /mnt/harddrive
/ # ls /mnt/harddrive
boot dev lib home lib misc opt sbin tmp
bindb etc initrd mnt proc sbin tmp
```

- Unlike Windows, there is only one filesystem – therefore you must mount to a mount point (i.e. empty directory) in the root filesystem
- Mounting a block driver into the filesystem gives access to the files on the device as a new directory
- Easy manipulation of flash, hard drive, compact flash and other storage media
- Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem
- The above example shows mounting the DM6446 DVEVM into an NFS root filesystem
The hard disc drive (HDD) on the DaVinci DM6446 DVEVM comes formatted with ext3 filesystem. This robust filesystem helps to prevent errors when the system is shut down unexpectedly (which happens quite often when developing embedded systems). Other filesystems include:

### MontaVista : Supported File Systems Types

**Harddrive File systems:**
- ext2: Common general-purpose filesystem
- ext3: Journaled filesystem - Similar to ext2, but more robust against unexpected power-down
- vfat: Windows "File Allocation Table" filesystem

**Memory File systems:**
- jffs2: Journaling flash filesystem (NOR flash)
- yaffs: yet another flash filesystem (NAND flash)
- ramfs: Filesystem for RAM
- cramfs: Compressed RAM filesystem

**Network File systems:**
- nfs: Share a remote linux filesystem
- smbfs: Share a remote Windows® filesystem

---

**Accessing Files**

Manipulating files from within user programs is as simple as...

- myFileFd = fopen("/mnt/harddrive/myfile","rw");
- fread ( aMyBuf, sizeof(int), len, myFileFd );
- fwrite( aMyBuf, sizeof(int), len, myFileFd );
- fclose( myFileFd );

Additionally, use fprintf and fscanf for more feature-rich file read and write capability
Using Character Device Drivers

Simple drivers use the same format as files:

- `soundFd = open("/dev/dsp", O_RDWR);`
- `read (soundFd, aMyBuf, len);`  \[ len always in # of bytes \]
- `write (soundFd, aMyBuf, len );`
- `close (soundFd );`

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

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

Notes:
- More complex drivers, such as V4L2 and FBDEV video drivers, have special requirements and typically use ioctl commands to perform reads and writes
- `/dev/dsp` refers to the “digital sound processing” driver, not the C64x+ DSP
Linux OSS (audio) Driver

OSS Driver consists of two main parts
- Mixer – allows for one or more inputs
- Sound device

Example – Audio Capture

Note: This is the example found in Lab 6a
main.c

```c
int main(int argc, char *argv[])
{
    int status = EXIT_SUCCESS;
    void *audioThreadReturn;

    /* Set the signal callback for Ctrl-C */
    signal(SIGINT, signal_handler);

    /* Call audio thread function */
    audioThreadReturn = audio_thread_fxn((void *) &audio_env);

    if( audioThreadReturn == AUDIO_THREAD_FAILURE )
    {
        DBG("audio thread exited with FAILURE status\n");
        status = EXIT_FAILURE;
    }
    else
    {
        DBG("audio thread exited with SUCCESS status\n");
    }

    exit(status);
}
```

/* Callback called when SIGINT is sent to the process (Ctrl-C). */

```c
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting\n");
    audio_env.quit = 1; // used as while loop condition
}
```
Driver Documentation

While the following screen capture is from an earlier revision of the DVSDK, the driver documentation in the current revision is located similarly; that is, it’s in the docs folder inside the PSP module.

The PSP (Platform Support Package) is a name unique to TI. The PSP team inside TI is responsible for the drivers and Linux kernel support. While TI uses the name PSP, other TI 3rd party Linux vendors name their packages: LSP (Linux Support Package), or BSP (Board Support Package). Most of these 3rd Party Linux vendors pull their original driver code from TI’s PSP content, although they may very well change it as they integrate it into their distributions. And, the beauty of open source development is that any improvements they make should end up being reflected in the common source-code GIT tree.
Lab 6 - Using the OSS Driver

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 OSS driver as well as basic file I/O. Lab portions 06a and 06b are inspection labs. In lab06a you will analyze the function calls necessary to record audio from line input to a file. In lab06b you will analyze the function calls necessary to playback audio from this recorded file to the driver.

In lab06c, you will combine labs 06a and 06b into a single application that loops the audio through without recording to a file. For an extra challenge, advanced students may want to try to build lab06c without referring to the procedure.

Lab 6 – Using the Linux OSS Driver

These labs demonstrate the Linux OSS driver as well as basic file I/O.

Lab 6a: Audio Record
Lab 6b: Audio Play Back
Lab 6c: Audio Loop-thru

Notes:

◆ Labs 6a and 6b:
  • We have provided the necessary files.
  • You need to build them using the gMake tool
  • Explore the audio recorder and playback, respectively.

◆ Lab 6c:
  • You will use the record and playback applications as a template for creating an audio loop-thru application.
Lab6a_audio_record

**Goal:** Analyze the function calls necessary to record audio from a line input to a file.
- Inspection lab only.
  1. Inspect the source files in this application.
  2. Increase the volume of the audio (’cause we’ve got CHEAP speakers).
  3. Listen to the audio using: `cat /dev/dsp > /dev/dsp`
  4. Build and run the application: Result: capture audio into a file: `audio.raw`.
  5. Add a new DBG() statement and inspect how DBG/ERR macros work in the system.

**File Management**

1. Change to the `/home/user/workshop/lab06a_audio_record/app` directory.
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 audio_input_output.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:

- **audio_input_setup()**: opens and configures the audio input driver. This function uses `malloc()` to allocate a RAM buffer to store the audio input and uses `fopen()` to open a file (audio.raw) for recording.
- **while()**: will execute until the `envPtr->quit` global variable is set to true.
  - Inside the **while()** loop, `read()` is used to read data from the audio input driver (OSS) 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 at the top of the file. 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.

audio_input_output.c

This file contains helper functions used to setup and cleanup the audio input and audio output.

- audio_input_setup() configures the /dev/dsp audio device and the /dev/mixer device which handles mixing of audio inputs into the driver. (In the current implementation, the mixer is configured to take the line input from the DVEVM. Microphone input is also a viable option).

- The functions which are available for use are audio_input_setup(), audio_input_cleanup(), audio_output_setup() and audio_output_cleanup(). There are two more functions – init_mixer() and init_sound_device(), which are private functions used by the other four.

Build and Run the application

4. Increase audio volume.

We found that many un-amplified speakers (like those built into the small LCD monitors used in many of our workshop classrooms) make it hard to hear the audio. To help rectify this, we recommend adding the following two lines to the /dev/mixer setup code in the file audio_input_output.c – in the function init_mixer():

```c
int volume = 100;
ioctl(mixerFd, SOUND_MIXER_WRITE_VOLUME, &volume);
/* Report mixer input settings and warn user if neither mic nor line */
```

5. Build and install the application using gMake, i.e. “make all install”.

6. Listen to the audio via the “cat” command (i.e. not using the application just built).

Open a terminal to the DVEVM board. Log in and use the following terminal command to test your audio connection:

```
cat /dev/dsp > /dev/dsp
```

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. This effectively tests our audio connections…now let’s see if our program works…
7. **Execute the .*/app_DEBUG.x470MV application.**

   Navigate to /opt/workshop on the DVEVM board.

   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
   ```

   Check to see if data was captured into the audio.raw file by checking the size and date/time in the temp directory:

   ```
   ls -l /tmp
   ```

   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.

**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 a fprintf() function.

9. **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");
   ```

10. **Build and run both debug and release profiles, 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 |

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

. What is the difference between DBG and ERR? _________________________________

. _______________________________________________________________________

12. 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.

AUDIO VOLUME – PLEASE READ

As a courtesy to all of the other students around you (as well as the instructors), please modify the volume control on the screens to a REASONABLE level – so you can hear it without straining – but not so loud that the people in a conference room IN THE NEXT BUILDING can hear it. Thanks. Everyone will certainly appreciate this kind gesture.
Lab06b_audio_playback

- **Goal:** Analyze the function calls necessary to play back audio from a recorded file to the driver.
- Inspection lab only.
  1. Inspect `audio_thread.c` and the associated helper functions.
  2. Build and run the application: Result: audio in audio.raw is sent to the audio driver.

---

**File Inspection**

13. In RedHat Linux (VMware PC), change to the directory:

```
/home/user/workshop/lab06b_audio_playback/app
```

14. 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:

- `audio_thread_fxn()` utilizes `audio_output_setup()` to:
  - `open` and configure the audio output driver
  - uses the `fopen()` function call to open a file for playback
  - The `malloc()` 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`)
  - `write()` method is used to write the data to the OSS driver.
  - When the `envPtr->quit` variable is set to true (occurs when the user presses Ctrl-C in the terminal), the loop exits
  - `audio_thread_fxn()` enters a “cleanup” phase (opposite of setup) before exiting.
Build and Run the Application

15. Build and **install** the application using gMake – i.e. “make debug install”.

16. Make sure that **audio.raw** was created properly.

   Navigate to `/opt/workshop` in the DVEVM 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 into the board’s filesystem via the nfs file sharing protocol.)

   If you have powered off or reset the DVEVM since running the `lab6a_audio_record` application, the `/tmp/audio.raw` file will have been cleared from RAM memory. If this is the case, run gMake again from the `lab06a_audio_record` directory to re-install the audio recorder. Run it once in either debug or release mode to record the `/tmp/audio.raw` file, and then return to `lab06b_audio_playback` and run gMake to re-install the playback utility.

17. Execute the `./app_DEBUG.x470MV` 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

1. Which ioctl command is called in the **init_mixer()** function of `audio_input_output.c` in order to specify whether the microphone or line input will provide the audio source for the driver?

   . ________________________________________________________________

   . ________________________________________________________________

2. Which ioctl commands set the number of channels and sample rate for the OSS driver (two different ioctl commands are needed), and in which function of `audio_input_output.c` are they found?

   . ________________________________________________________________

   . ________________________________________________________________
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 OSS input), copied from in? out buffer, then played back (to OSS output).
Lab 6 – Lab Flow Discussion

---

**lab06c_audio_loothru**

<table>
<thead>
<tr>
<th>lab06a_audio_record</th>
<th>lab06b_audioPlayback</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio_thread.c</td>
<td>audio_thread.c</td>
</tr>
<tr>
<td>OSS</td>
<td>audio.raw</td>
</tr>
<tr>
<td>fwrite()</td>
<td>fread()</td>
</tr>
</tbody>
</table>

**Lab 6a**
Which function gets an audio buffer?
Get data: __________________________
Put data: fwrite() inputBuffer ? outFile

**Lab 6b**
Which function puts to the OSS driver?
Get data: fread() inputFile ? outputBuffer
Put Data: __________________________

---

**lab06c_audio_loothru**

<table>
<thead>
<tr>
<th>lab06a_audio_record</th>
<th>lab06b_audioPlayback</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio_thread.c</td>
<td>audio_thread.c</td>
</tr>
<tr>
<td>OSS</td>
<td>audio.raw</td>
</tr>
<tr>
<td>fwrite()</td>
<td>fread()</td>
</tr>
</tbody>
</table>

**Lab 6a**
Which function gets an audio buffer?
Get data: read() inputFd ? inputBuffer
Put data: fwrite() inputBuffer ? outFile

**Lab 6b**
Which function puts to the OSS driver?
Get data: fread() inputFile ? outputBuffer
Put Data: write() outputBuffer ? outputFd
Lab 6a
Which function gets an audio buffer?
Get data: `read() inputFd ? inputBuffer`
Put data: `fwrite() inputBuffer ? outFile`

Lab 6b
Which function puts to the OSS driver?
Get data: `fread() inputFile ? outputBuffer`
Put data: `write() outputBuffer ? outputFd`

For Lab06c:
- Take the code from lab06b and copy to Lab06c.
- Replace the `fread()` in Lab06b with the `read()` from Lab06a.
Questions

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 OSS 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 OSS driver. Which function was used to GET (read) the audio data?

  GET audio data: __________________________________________

  PUT audio data: `write()` `outputBuffer` -> `outputFd` ________

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

  Get audio data: __________________________________________

  Put audio data: __________________________________________
**File Management**

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

```bash
cd /home/user/workshop
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 the file that differs between the two is `audio_thread.c`) we 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 correct.

---

**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` (we suggested this one in the lab05 instructions)
   - `emacs audio_thread.c`
   - `gvim audio_thread.c`

   If you are not already familiar either of these options, gedit is probably the most intuitive and user friendly option.

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 that which is needed for the OSS driver.)

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

   - `INPUT_OSS_INITIALIZED`
   - `INPUT_BUFFER_ALLOCATED`
   - `OUTPUT_OSS_INITIALIZED`
   - `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. Replace the input file initialization code with initialization code for the OSS audio input.
   - Remove the following code section in `audio_thread_fxn()` of `audio_thread.c`:
     ```c
     /* Open input file */
     ...
     
     /* Record that input OSS device was opened in initialization bitmask */
     ...
     ```
   - Replace with the initialization code for the OSS audio input.

   **Hint:** The simplest method is to copy and paste the OSS input channel initialization code from `audio_thread.c` in the lab06a_audio_record lab. Use the sections labeled:
     ```c
     // Open an OSS device channel for audio input
     // Record that input OSS device was opened in initialization bitmask
     // Create input buffer to read into from OSS input device
     // Record that the input buffer was allocated in initialization bitmask
     ```

23. Replace the file input code with the proper OSS audio input channel read command.
   - Locate the key `while()` loop in `audio_thread_fxn()` and remove the following section of code:
     ```c
     // Read buffer from audio file
     ```
   As with the previous step, you may want to copy the code that you need from the `audio_thread_fxn()` of lab06a_audio_record.

24. Create an audio pass-thru using `memcpy()` to copy data from the input to the output.
   - If you have completed the previous step correctly, the audio input driver is configured to fill the “inputBuffer” buffer, and the audio output driver is set up to take data from the “outputBuffer” buffer. Insert a `memcpy()` command to copy `inputBuffer` into `outputBuffer`.
   - You can use “man memcpy” in the Linux terminal for more information on `memcpy()`. This function is a little different than most in that it takes the destination buffer as the first argument then the source buffer as the second argument, so don’t be fooled!

   **Help with `memcpy()`:**
   - If you honestly can’t figure out how to write the `memcpy()` and you just want to move on to the next step, reference the solution for lab06c (`audio_thread.c` inside the `while` loop) and copy it into your `audio_thread.c` in the proper place. The goal here is to have success.
25. Replace the file cleanup code with the OSS cleanup functions.

- Locate the Thread Delete Phase after the “cleanup:” tag in the audio_thread_fxn() of audio_thread.c. Remove the code section labeled:

  /* Close input file */

- Replace the file cleanup code you removed with the proper OSS audio input channel cleanup functions. (Again, the simplest method is to copy and paste the OSS input channel cleanup code from audio_thread.c found in lab06a_audio_record.)

26. You can now close audio_thread.c in lab06a_audio_record.

   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.

27. 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:

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

   The following must be added:

   ```c
   int   inputFd = 0;
   char  *inputBuffer = NULL;
   int   vol = LEFT_GAIN | (RIGHT_GAIN << 8);
   ```

28. Also, a few #defines are required for some new constants that were added.

   You will also need to #define LEFT_GAIN and RIGHT_GAIN at the top of the file. Use the maximum gain of 100 (decimal) for each.

   ```c
   #define LEFT_GAIN 100
   #define RIGHT_GAIN 100
   #define MIXER_DEVICE "/dev/mixer"
   ```

   Save and close audio_thread.c

29. Using gMake, build the executable using the debug and install rules.

30. Go to the /opt/workshop folder on the target and run the application.

   You should hear audio playing.

   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.
(Optional) Lab06d_challenge1

Note: Before trying this challenge, go ahead and make a Lab06d_challenge1 directory and copy (recursively) your Lab06c_audio_loopthru to it.

Take the loopback application of Lab06c_audio_loopthru and modify the audio thread to add reverb. Reverb is accomplished by delaying the audio input by a fixed amount (usually between 1/8 to 1/2 of a second) and adding the delayed input, attenuated by some factor (usually 0.4 to 0.7) back into undelayed input stream:

\[
\begin{array}{c}
\text{+} \\
\text{z-1} \\
\text{x} \\
\text{const}
\end{array}
\]

The delay is accomplished with a special buffer called a delay line. After each input buffer is processed, the items in the delay line buffer are shifted downward by one input buffer size, and the input buffer is placed at the top. The reverb values that are added back into the stream come from the bottom of the delay line, and the size of the delay line determines the amount of delay.

The solution for this challenge is at solutions/lab06d_challenge1. All reverb mechanics have been placed within the while loop of audio_thread_fxn for you to examine. Try playing with the reverb attenuation and delay to generate different effects.
Video Driver Details

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).

Outline

- V4L2 Capture Driver
  - Using mmap
  - V4L2 Coding
- FBDev Driver
  - Video Planes
  - FBDev Coding
- Video Display Boot Arguments
- Lab Exercise
  - Video OSD
  - Video Recording
  - Video Playback
  - Video Loopthru
Chapter Topics

**Video Driver Details**

- **v4L2 Capture Driver** ................................................................. 7-3
  - Overview ................................................................................. 7-3
  - How v4L2 Works .................................................................... 7-4
  - Using mmap ............................................................................ 7-6
  - v4L2 Coding .......................................................................... 7-8

- **FBdev – Display Driver** .............................................................. 7-9
  - Overview ................................................................................. 7-9
  - Video Planes .......................................................................... 7-9
  - Using Pan feature for Ping-Pong Buffers .................................... 7-11
  - FBdev Coding ......................................................................... 7-12

- **Setting Video Display Properties** ............................................ 7-13

- **Lab 7 – Using Video Drivers** ....................................................... 7-14
  - Lab07a_osd_setup ................................................................. 7-14
  - Lab07b_video_record ............................................................. 7-20
  - Lab07c_video_playback .......................................................... 7-23
  - Lab07d_video_loopthru ............................................................ 7-24
  - Lab07e_challenge1 — Scrolling Banner Challenge ....................... 7-32
v4L2 Capture Driver

Overview

v4l2 Driver Overview

- v4l2 is a standard Linux video driver used in many Linux systems
- Supports *video input* and *output*
  - In this workshop we use it for video input only
- Device node
  - Node name: /dev/video0
  - Uses major number 81
- v4l2 spec: [http://bytesex.org/v4l](http://bytesex.org/v4l)
- Driver source location:
  .../lsp/ti-davinci/drivers/media/davinci vpfe.c
How v4L2 Works

v4L2 Driver Queue Structure

- Application takes ownership of a full video buffer from the outgoing driver queue using the VIDIOC_DQBUF ioctl
- After using the buffer, application returns ownership of the buffer to the driver by using VIDIOC_QBUF ioctl to place it on the incoming queue

v4L2 Enqueue and Dequeue

- Buffers typically exist in driver's memory space
- The dequeue call makes the data available to the app
**v4l2 Enqueue and Dequeue**

- Buffers typically exist in driver’s memory space.
- The dequeue call makes the buffer available to the app.
- Even after DQ, buffers still exist in the driver’s memory space but not the application’s.

**v4l2 Enqueue and Dequeue**

- Buffers typically exist in driver’s memory space.
- The dequeue call makes the buffer available to the app.
- Even after DQ, buffers still exist in the driver’s memory space but not the application’s.
- Read and write operations copy buffers from driver’s memory space to app’s or vice-versa.
Using mmap

**mmap – A Better Way**

- Standard read and write copy data from driver buffer to a new buffer in application process’s memory space
- Use mmap to expand the application process’s memory space to include the driver buffer
- Returns a pointer to the location in the app’s memory space

**v4L2 Queue Synchronization**

- Driver fills IQ buf and places on OQ
- The VIDIOC_DQBUF ioctl blocks the thread’s execution waits if until a buffer is available on the output queue
- When driver adds a new, full buffer to the output queue, the application process is released
- Dequeue call completes and application resumes with the following set of commands
v4L2 Queue Synchronization

- The VIDIOC_DQBUF ioctl blocks the thread’s execution waits until a buffer is available on the output queue.
- When the driver adds a new, full buffer to the output queue, the application process is released.
- Dequeue call completes and application resumes with the following set of commands.
v4L2 Coding

**v4L2 Buffer Passing Procedure**

```c
while(cond){
    ioctl(v4l2_input_fd, VIDIOC_DQBUF, &buf);
    bufPtr = mmap(NULL, buf.length, PROT_READ | PROT_WRITE, MAP_SHARED, v4l2_input_fd, buf.m.offset);
    doSomething(bufPtr, buf.length, ...);
    munmap(bufPtr, buf.length);
    ioctl(v4l2_input_fd, VIDIOC_QBUF, &buf);
}
```

- A simple flow would be: (1) DQBUF the buffer, (2) map it into user space, (3) use the buffer, (4) unmap it, (5) put it back on the driver's queue
- More efficient code would map each driver buffer once during initialization, instead of mapping and unmapping within the loop
- Alternatively, later versions of the driver allow you to create the buffer in 'user' space and pass it to the driver

---

**Commonly Used v4L2 ioctl’s**

**Data Structures**

```c
struct v4l2_requestbuffers req;
req.count; // how many buffers to request
req.type; // capture, output, overlay
req.memory; // mmap, userptr, overlay

struct v4l2_buffer buf;
buf.index; // which driver buffer
buf.type; // matches req.type
buf.memory; // matches req.memory
buf.m.offset; // location of buffer in driver mem
```

- Request the driver allocate a new buffer
  ```c
  ioctl(fd, VIDIOC_REQBUFS, &req);
  ```

- Get information on a driver buffer
  ```c
  ioctl(fd, VIDIOC_QUERYBUF, &buf);
  ```

- Enqueue and Dequeue buffers to/from driver
  ```c
  ioctl(fd, VIDIOC_QBUF, &buf);
  ioctl(fd, VIDIOC_DQBUF, &buf);
  ```
FBdev – Display Driver

Overview

FBdev Driver Overview

- FBdev is a standard Linux video output driver used in many Linux systems
- Can be used to map the frame buffer of a display device into user space
- Device nodes have major number 29
- Device nodes have a minor number x
- Uses /dev/fb/x node naming convention

Video Planes

Multiple Video/OSD Windows

Source: DirectFBOverview.pdf
Video Port Back End Features

- Two video windows for picture-in-picture
- Two OSD windows or one OSD window + attribute window
- /dev/fb2 attribute mode provides pixel-level alpha blending

OSD Attribute Window

- Allows Pixel by Pixel Blending of OSD0 and Video Windows
- Uses a 4-bit, Bit-map Window

Blinking

8-level blending

000: 00.0%, 100% Video
001: 12.5%, 87.5% Video
010: 25.0%, 75.0% Video
...
110: 75.0%, 25.0% Video
111: 100%, 00.0% Video
Using Pan feature for Ping-Pong Buffers

For Each /dev/fb/x Video Plane

- Named FBdev because it gives direct access to the display device’s frame buffer
- FBIO_PAN_DISPLAY allows users to pan the active display region within a virtual display buffer

Ping-pong Buffers with FBdev

- FBdev has no video buffer queue (provides direct access to the display device’s frame buffer)
- Use FBIO_PAN_DISPLAY to switch between 2 or more buffers in the virtual buffer space
- Use FBIO_WAITFORVSYNC to block process until current buffer scan completes, then switch.
FBdev Coding

Buffer Synchronization

```c
pVirtualBuf = mmap(NULL, display_size * NUM_BUFS,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED,
                     FbdevFd, 0);

while(cond){
    // map next frame from virtual buffer
    display_index = (display_index + 1) % NUM_BUFS;
    ioctl(pFbdevFd, FBIOGET_VSCREENINFO, &vinfo);
    vinfo.yoffset = vinfo.yres * display_index;
    // write pixels into next video frame
    genPicture(...);
    // switch to next frame
    ioctl(pFbdevFd, FBIOPAN_DISPLAY, &vinfo);
    // wait for current frame to complete
    ioctl(pFbdevFd, FBIO_WAITFORVSYNC, NULL);
}
```

Commonly Used FBdev ioctl

Data Structures

```c
struct fb_fix_screeninfo myFixScreenInfo;
myFixScreenInfo.smem_len;     // length of framebuffer

struct fb_var_screeninfo myVarScreenInfo;
myVarScreenInfo.xres;         // visible pic resolution
myVarScreenInfo.xres_virtual; // virtual pic resolution
myVarScreenInfo.xoffset;      // from virtual to vis
```

Get or put variable screen information

```c
ioctl(fd, FBIOGET_VSCREENINFO, &myVarScreenInfo);
ioctl(fd, FBIOPUT_VSCREENINFO, &myVarScreenInfo);
```

Get fixed screen information

```c
ioctl(fd, FBIOGET_FSCREENINFO, &myFixScreenInfo);
```

We use Pan to switch output buffers

```c
ioctl(fd, FBIOPAN_DISPLAY, &myVarScreenInfo);
```

After writing buffer and pan_display, wait for current to finish

```c
ioctl(fd, FBIO_WAITFORVSYNC, NULL); // arg 3 is not used
```
Setting Video Display Properties

Video Display Driver Properties (1)

- Set resolution, bit-depth, size-of-buffer, and offset for FBdev’s OSD and video drivers
- Video display properties can be set in one of 3 ways:

1. **Linux boot arguments** *(examine bootargs as set in Lab 4)*
   
   ex: video=davincifb:osd0=720x480x16,1350K:osd1=720x480,1350K:vid0=720x480,2025K:…

2. **From the command-line shell**
   
   example: fbset -fb /dev/fb/1 -xres 720 -yres 480 -vxres 720 -vyres 1440 -depth 16 -nonsid 1

3. **In your program** *(look in video_output.c and video_osd.c)*
   
   examples:
   ```
   vidInfo.xres = 720;
   attrInfo.xres_virtual = 768;
   ```

* Note: Default properties are OK for all settings except xres_virtual, which now defaults to 720, but must be set to 768 for NTSC/PAL. (Small migration issue, as the previous driver defaulted to 768.)

---

Video Display Driver Properties (2)

The user can choose the following, as well:

- **Display mode**: NTSC, PAL
- **Display type**: Composite, S-video, Component

Set mode and type with one of three ways:

1. **Linux boot arguments** *(examine bootargs set in Lab 4)*
   
   example: davinci_enc_mngr.ch0_output=COMPOSITE
davinci_enc_mngr.ch0_mode=NTSC:…

2. **Command-line** *(modifying /sysfs)*
   
   example: Write appropriate value to text file found at either:
   ```
   /sys/class/davinci_display/ch0/output
   /sys/class/davinci_display/ch0/mode
   ```

3. **In your program**
   
   ex:
   ```
   fd = fopen("/sys/class/davinci_display/ch0/output",O_RDWR);
   write(fd,"COMPOSITE",10)
   ```
Lab 7 – Using Video Drivers

Lab 7 is composed of 4 parts:

- **Lab 07a:** You will build an **on-screen display** for the DM644x using the FBDEV driver – INSPECTION LAB only.

- **Lab 07b:** Examines v4L2 video capture via a simple **video recorder** application – INSPECTION LAB only.

- **Lab 07c:** Examines the FBdev display driver using via a **video display** application. This application plays back the file recorded in lab 07b, as well as adding the OSD from lab 07a – INSPECTION LAB only.

- **Lab 07d:** You will combine the recorder and player applications into a **video loop-thru** application, respectively.

**Lab07a_osd_setup**

- **Goal:** to build an **on-screen display** for the DM644x using the FBDEV driver.
- **From a coding perspective, it’s an inspection lab only.**

1. Create your own **custom picture** for the OSD window (using gimp), saving the picture to 16-bit format RGB565 (as osd.r16).
2. Inspect **video_thread.c** and helper functions (inside video_osd.c).
3. **Build, run.** Result: see your customer banner displayed on screen (no video yet...).
Lab 07a Procedure

1. In Ubuntu Linux, change to the directory:
   
   /home/user/workshop/lab07a_osd_setup/osdfiles

2. Explanation of bmpToRgb16.x86U.

   There is one file in the /osdfiles directory – an executable named: bmpToRgb16.x86U

   This is a simple utility convert’s 24-bit RGB bitmaps to 16-bit RGB bitmaps. (That is, from
   RGB 888 to RGB 565.) This was needed since paint programs only create the former, while
   the OSD hardware requires the latter 16 bpp format.

   The source code for the bmpToRgb16.x86U utility is in the following directory:
   
   ~/workshop/lab00b_osd_utility/app

3. Open the Gimp (open-source) paint program by typing “gimp” in the terminal.

4. Create a customer banner picture.

   Create a new file using: File \ New

   Set the height and width to 60 x 720, 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: 720 pixels
   Height: 60 pixels

5. 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
     drag the mouse over the 720x60 image
     area.
   – Add text or paint something over the
     gradient with either of these tools,
     respectively.

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

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

   Add text by selecting the
   font icon is the capital T

   Paint brush
6. **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. It doesn’t matter what you name the file, but `osd.r16` will make it easy to remember it’s a 16-bit file.

![Gimp Save Image](image)

You will also need to specify the R5 G6 R5 format that’s required for the OSD hardware.

7. **Copy your file to 16-bit format file to /opt/workshop.**

   ```bash
   cp osd.r16 ~/workdir/filesys/opt/workshop
   ```

8. **Change to the lab07a_osd_setup/app directory and list the contents.**
9. Examine two of the video files.

**video_osd.c**

`video_osd.c` contains two helper functions:

- `video_osd_setup()`
- `video_osd_cleanup()`

These functions are used to open and configure the OSD and attribute windows.

Three other helper functions are provided:

- `video_osd_place()`: places a picture on the OSD display and modifies the attribute window with the transparency value provided.
- `video_osd_scroll()`: a more complex version 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.

**video_thread.c**

The main function in `video_thread.c` is `video_thread_fxn()` which uses the helper functions described above to do the following:

- calls `video_osd_setup()` to open the OSD and the attribute windows. These windows are mmap’ed into the application space and their locations are returned through the osdDisplay and attrDisplay pointers. Likewise, the file descriptors for these devices are returned through osdFd and attrFd.
- calls `video_osd_circframe()` to place a white background frame around the video output (0xFFFF is two white pixels in 5-6-5 RGB encoding) and blend this frame about 50-50 with the video window (specified with the 0x44 transparency value).
- reads the custom banner picture you created earlier and stores it into a picture buffer. This picture buffer is then passed to `video_osd_place()`, which places the picture on the OSD window with a transparency of 0x77 (no transparency) and a y offset of 360 (near the bottom of the screen).

The application assumes the picture is supplied in a 720 x 60 16-bit RGB (5-6-5) format, which should be the case if you followed the previous gimp instructions.
10. **Build and install the application.**

11. **Execute the application on the target.**

    target# /opt/workshop/app_DEBUG.x470MV

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

---

**Note:** If your video screen is blank after running your program, one quick test to make sure it isn’t just the attribute buffer that is incorrect, try the following:

    Ubuntu# cp ~/workshop/lab00c_attr_files/* ~/workdir/filesys/opt/workshop

    Dvevm# cat allosd.attr > /dev/fb/2

    Rather than typing the `cat` command, you can use the `apply_attr.sh` command:

    Dvevm# ./apply_attr.sh allosd.attr

---
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. Why is it necessary to run the bmpToRgb16.x86U utility on the bitmapped image that you created with gimp before it could be displayed in the DM6446 on-screen-display window?

   -
   -
   -
Lab07b_video_record

- **Goal:** Examine v4L2 video capture via a simple video recorder app.
- **Inspection lab only.**
  1. Examine helper functions (setup, cleanup, wait_for_frame) in video_input.c.
  2. Examine `video_thread_fxn()` in video_thread.c.
  3. Examine `main.c` (how the signal handler is created/used, then calls `video_thread_fxn`).
  4. **Build, run.** Result: create a file (video.raw) that contains about 2s of captured video.

13. Change to the directory:

   `/home/user/workshop/lab07b_video_record/app`

14. Examine the video files:

   **video_input.c**

   This file contains helper functions used to setup and cleanup the video input device driver.

   The functions contained in `video_input.c` are:
   - `video_input_setup()`: opens and initializes the video window device as well as setting the attribute window to transparent.
   - `video_output_cleanup()`: frees the video window device.
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:
  - video_input_setup(): opens and configures the V4L2 video input driver and uses the fopen function call to open a file for recording.
  - while() loop:
    - Executes until the envPtr->quit global variable is set to true.
    - VIDIOC_DQBUF ioctl dequeues the video frame from the V4L2 driver (processing blocks/pauses if buffer is not available, yet).
    - The video frame is copied into the file. We perform this using two fwrite’s:
      1. It writes the size of the captured buffer to the file; followed by,
      2. the actual video data from our capture buffer.
    - VIDIOC_QBUF ioctl is used to enqueue the video frame back to the V4L2 driver’s incoming (empty) queue.

When the envPtr->quit variable is set to true (occurs in our signal_handler function when the user presses Ctrl-C in the terminal) this video capture loop exits and the application proceeds to the cleanup phase, before exiting.

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.

15. Once again, if you use PAL formatted video in your region, you need to edit one of the v4l2 ioctl commands.

We need to pass the video standard to the v4l2 driver. In this case, we put in the correct value for PAL format, you just need to comment out the NTSC line and uncomment the PAL.

```c
/* Set analog video input standard to either NTSC or PAL */
std = V4L2_STD_NTSC; // Comment out this line
// std = V4L2_STD_PAL; // Uncomment this line
if(ioctl(captureFd, VIDIOC_S_STD, &std) == -1) {
```

16. Build and install the application using gMake using “make debug install”.

This step is for PAL format regions only
17. **Run the application on the target … then stop using `ctrl-c`.**

Open a terminal to the DVEVM board. Navigate to the target’s `/opt/workshop` directory, and then execute the `.app_DEBUG.x470MV` application. Finally, press `ctrl-c` to exit the 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 there was an **error** writing to the file after about 2 seconds, and the application will exit (unless you press `ctrl-c` before then). Check the following to ensure that the video has recorded properly:

```
ls -lsa /tmp/video.raw
```

The file should be about 60MB in size. The reason that the application only records for two seconds is that after this time, the raw video stream that is being recorded will use the entire 60MB of RAM that is available in 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

18. Change to the directory:

/home/user/workshop/lab07c_video_playback

19. Copy the osdfiles directory from lab07a to lab07c.

mkdir -p osdfiles
cd osdfiles
cp -R -f ~/workshop/lab07a_osd_setup/osdfiles/* .
cd ../app  (so you should be in the ~/workshop/lab07c_video_playback/app directory)

20. Examine video_output.c:

This file contains helper functions used to setup and cleanup the video output device driver:

- `video_output_setup()` opens the video display driver and accesses the video display configuration information and modifies the virtual resolution settings, whose default values are incorrect for our needs. Next, it `mmap`'s the video buffers that’ll be used.

- `flip_display_buffers()` exchanges the working video buffer with the currently displayed video buffer.

21. Build and install the application.

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

Navigate to `/tmp` in the DVEVM 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 DVEVM since running the `lab07b_video_record` application, the `video.raw` file will have been cleared from RAM memory. If so, go back and build/install `lab07b_video_record` to create the `video.raw` file again.
23. Execute the ./app_DEBUG.x470MV 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 and then exit when you press Ctrl-C.

**Lab07d_video_loopthru**

In this portion of the lab, you will combine the lab07b_video_record 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 the FBdev output display driver (along with your customized OSD banner from Part A).

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). It is most likely obvious that we’ll need to get rid of the “file reads/writes” and replace them with some type of copy of the input video buffer to the output video buffer.

---

**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_record**, 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?

<table>
<thead>
<tr>
<th>GET video data:</th>
<th>PUT video data:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ________________________________</td>
<td>1. fwrite() frame size ____________________</td>
</tr>
<tr>
<td>2. fwrite() the video frame __________</td>
<td>2. fwrite() the video frame __________</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>GET video data:</th>
<th>PUT video data:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ________________________________</td>
<td>1. write frame to working buffer __________</td>
</tr>
<tr>
<td>2. ________________________________</td>
<td>2. flip buffers __________________________</td>
</tr>
</tbody>
</table>

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**?

<table>
<thead>
<tr>
<th>Get video data:</th>
<th>Put video data:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ________________________________</td>
<td>1. ________________________________</td>
</tr>
<tr>
<td>2. ________________________________</td>
<td>2. ________________________________</td>
</tr>
</tbody>
</table>
To summarize, the following lab procedure will take the record and playback files and combine them into a loopthru example.

**Lab 7b**

What functions "get" a video frame?

Get data:
1. dequeue buffer

Put data:
1. fwrite() frame size
2. fwrite() the video frame

**Lab 7c**

What functions "display" a frame?

Get data:
1. fread() frame size
2. fread() the video frame

Put data:
1. write frame
2. flip buffers

For Lab07d:
- Take the code from lab07c
- Replace the fread's with the v4l2 capture code found in lab 07b

**Lab 7d**

For Lab07d:
- Take the code from lab07c
- Replace the fread's with the v4l2 capture code found in lab 07b

---

**Note:** For those advanced students who would like a challenge, see if you can accomplish this lab without referring to the procedure below.
**Lab07d Procedure**

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

   ```bash
   cp -R -f ~/workshop/lab07c_video_playback/* ~/workshop/lab07d_video_loopthru
   cd ~/workshop/lab07d_video_loopthru/app
   
   22. Add the video input files to your project.
   
   The lab files we just copied (from `lab07c_video_playback`) only contain the code to playback video. Now, we need to add the appropriate video input files by copying them from our `lab07b_video_record/app` folder.

   Copy the following two files from `lab07b_video_record/app` to the `lab07d_video_loopthru/app` project folder.

   ```
   video_input.c
   video_input.h
   ```

23. Reference the header file you just copied.

   Open `lab07d_video_loopthru/app/video_thread.c` for editing; then add a `#include` reference to the `video_input.h` under the comment banner:

```
    //** Application header files *
```

24. Change the definitions for video input.

   Since our loopthru app won’t need to read video from a file any longer, we must change the definitions for the video input. This requires two steps: remove the old, create the new. In `video_thread.c`:

   - Remove the `#define` constant declaration for `INPUTFILE`. (Don’t remove the `PICTUREFILE` definition as we still want to have this banner displayed on the OSD.)

```
    //** Input file *
    //#define_INPUTFILE    "/tmp/video.raw"
    #define PICTUREFILE   "osd.r16"
```

   - Create a `#define` constant for the location of the V4L2_DEVICE as a constant. By location, we are implying the node name (or virtual file name) for the v4l2 video input device.

```
   .  //** Video capture and display devices used *
   .  #define FBVID_OSD      "/dev/fb/0"
   .  #define FBVID_ATTR     "/dev/fb/2"
   .  #define FBVID_VIDEO    "/dev/fb/3"
   .  #define V4L2_DEVICE    " ___________________________
```

**Hint:** If you do not remember the name of this device, refer to the `video_thread.c` file in the `lab07b_video_record` directory.
25. Define that we want three capture buffers.

Similar to that for display, we need to indicate to our thread the number of capture buffers used in the input driver using the NUM_CAP_BUFS constant.

```c
/* Double-buffered display, triple-buffered capture */
#define NUM_DISP_BUFS 2
#define NUM_CAP_BUFS  3
```

26. 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_record` directory.

We suggest to cut/paste these variables from `lab07b_video_record/video_thread.c`.

```c
int  captureFd = 0;
VideoBuffer  *vidBufs = NULL;
unsigned int  numVidBufs = NUM_CAP_BUFS;
int  captureWidth;
int  captureHeight;
int  captureSize = 0;
struct v4l2_buffer  v4l2buf;
```

**Note:** `v4l2` above has an “el” (lowercase L) character, not a one!

27. Delete the code to open our input file.

Since we now need to read data directly from the video capture port – i.e. v4L2, 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:

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

```c
DBG("Opened file %s with FILE pointer %p\n", INPUTFILE, inputFile);
```
28. 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, 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

29. Replace the section that you removed in step 27 with the section from lab07b_video_record/app/video_thread.c that initializes the capture device.

This is the section of code that starts at the comment banner:

// Initialize the video capture device

and ends with the code under the comment banner:

// Record that capture device was opened in initialization bitmask.
30. 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 two fread() statements – that together read a video frame from the INPUTFILE – with the code needed to capture the frame from the v4l2 device. In this step, we’ll replace the first fread() function; in step 31, we will remove the second fread() function.

Replace:

```c
// Read size of next video frame to be read from video file
if( fread( &captureSize, sizeof( captureSize ), 1, inputFile ) < 1 )
    break;
```

with the code required to read from the v4l2 device. Again, it’s probably easiest to cut/paste this from lab07b_video_record – look for the code following these two banners:

```c
// Initialize v4l2buf buffer for DQBUF call
and
// Dequeue a frame buffer from the capture device driver
```

Please note, you’ll need to copy three both parts of this section:
- v4l2 buffer init (consisting of 3 lines of code)
- wait_for_frame() function  (We also used to call wait_for_frame(), but we now use the ‘blocking’ version of the DQBUF call, which means we don’t need to check first if a new frame is available.)
- DQBUF ioctl

**Hint:** Leave in the line that reads:  dst = displays[workingIdx];
31. 

**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.)

The second fread() function just happens to be located where we need to put our memcpy() call. Therefore, replace the fread() call:

```c
if( fread( dst, sizeof( char ), captureSize, inputFile ) < captureSize )
    break;
```

with a call to `memcpy(d, s, l)`. What three arguments should we use for `memcpy()`:

- **Destination:** ______________________________
- **Source:** ______________________________
- **Length:** ______________________________

**Hint:** To figure out the destination and size, just look at the fread() function we’re replacing in this step.

**Hint:** Admittedly, the source argument is more difficult to figure out. It might help to remember that we use an array of 3 capture buffers. (And if that isn’t enough of a hint, which array got filled-in by calling the `video_input_setup` function?)

Of course, once you figure out the name of the capture buffers array, you’ll need to figure out which item in the array to use. The simple answer is that the index of our buffer just happens to be stored in the v4l2 buffer data structure. (The one returned from DQBUF.)

OK, if you’ve been able to figure out the name of the buffer array, and then its index, you just need to find the starting address of the buffer. Luckily, this is pretty easy. The header file (video_input.h) provides the definition for the `VideoBuffer` data structure used by our buffer array. Once you look up the data structure’s definition, the element you want to use should be obvious.

**Hint:** If you’re still stumped, on the source argument, turn to the next page and see if you can make sense of the answer.
If you figured it out correctly, the memcpy() function should read:

```c
memcpy(dst, vidBufs[v4l2buf.index].start, captureSize);
```

32. QBUF?

Don’t forget to put the buffer back onto the queue when you’re done with it. This is one of the most common mistakes. If you don’t put the buffer back, the driver will never fill it up again. (Once again, checkout the `lab07b_video_record` file if you don’t remember how to code the QBUF ioctl.)

33. Cleanup the video input … rather than reading in from a file.

Finally, replace the section in the cleanup that closes the raw video input file with the corresponding cleanup code from `lab07b_video_record` that cleans up the capture driver.

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

Note: you should just close `video_thread.c` from `lab07b_video_record`, because you should not have made any changes to this file.

35. Build and install the application (using the debug rule)...then run it.

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

**Lab07e_challenge1 — Scrolling Banner Challenge**

36. Make a copy of the your results before we try the challenge.

Begin by copying the working `lab07d_video_loopthru` application into the `lab07e_challenge1` directory. Don’t forget to make a folder first, before copying the folder.

```
       cd ~/workshop
       mkdir -p lab07e_challenge1
       cp -R -f lab07d_video_loopthru/* lab07e_challenge1
```

37. Your challenge is to utilize the `video_osd_scroll` function provided in `video_osd.c` to not only place the banner you created in gimp, but scroll it either horizontally or vertically.

The basic procedure is to modify `video_thread_fxn` within `video_thread.c` to add a call to `video_osd_scroll`. Note that outside of the while loop, you must call `video_osd_place` once to initialize the banner region. (This function call sets the alpha blending for the banner region, whereas the scrolling function does not.)

Inside the while loop, you will need to maintain a variable which increments each iteration of the loop and represents either the x or y scrolling offset. Don’t forget to check this against the picture dimensions (720 pixel width and 60 pixel height) and reset back to zero before the scrolling exceeds the picture size. The `video_osd_scroll` function you are using does not check to make sure that the scrolling value is within the valid range, and bad things will happen if you don’t do this check!
Multi-Threaded Applications

Introduction

In this chapter an introduction to Linux will be provided. Those currently using Linux should already be familiar with the information presented here. Anyone new to Linux should find this foundational information helpful in providing context for concepts that come in later chapters.

Learning Objectives

At the conclusion of this chapter, you should be familiar with the basics of:

- What are Linux processes – and how are they like processors
- What are Linux threads – and how do they differ from processes
- How does Linux implement thread synchronization
- Using Linux real-time thread scheduling
Chapter Topics

Multi-Threaded Applications ........................................................................................................... 8-1

Linux Processes ................................................................................................................................. 8-3
Linux Threads ................................................................................................................................. 8-8
Thread Synchronization ...................................................................................................................... 8-12
Using Real-Time Threads .................................................................................................................. 8-15
Lab 8 ............................................................................................................................................... 8-21
Lab08a – Run Audio and Video in Separate Processes ................................................................. 8-21
Lab08b_audio_video ......................................................................................................................... 8-23
Lab08c_audio_video_rtime ............................................................................................................... 8-27
Lab08d_challenge1 -- Control Thread ............................................................................................ 8-28
Chapter 8 Appendix ......................................................................................................................... 8-29
Sidebar - Looking at the pthread arguments in detail: ................................................................. 8-29
Sidebar to the Sidebar – The devilish details of the pthread_create() 3rd arg ......................... 8-30
Linux Processes

### What is Our Main Goal?

- **Goal:** Run video and audio at the same time
- **What are different ways to accomplish this?**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1    | Two processors  
  - Lots of real estate and memory, costly  
  - How do you sync A/V? |
| 2    | Two processors (cmd line)  
  - Cmd line: ./AUD.x470MV & ./VID.x470MV  
  - How do you sync A/V? |
| 3    | Two processes (programatic)  
  - Start 2nd process programatically  
  - Memory protection (MMU)  
  - Context switch is difficult  
  - How do you sync A/V? |
| 4    | Two pThreads  
  - Uses fewer resources, faster  
  - pThreads (lightweight threads)  
  - Can share global variables |

### What is a Processor?

<table>
<thead>
<tr>
<th>Processor A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor B</td>
</tr>
<tr>
<td>Processor C</td>
</tr>
</tbody>
</table>

**System #1**

- Processor A  
  - Memory  
  - main(){  
    func1();  
    func2();  
    ...  
  }
- Processor B  
  - Memory  
  - main(){  
    func3();  
    func4();  
    ...  
  }
- Processor C  
  - Memory  
  - main(){  
    func5();  
    func6();  
    ...  
  }

Let's compare/contrast each of these...
What is a Process?

System #2

Processor A

Process A

Memory

main()
{
func1();
func2();
...}

Process B

Memory

main()
{
func3();
func4();
...}

Process C

Memory

main()
{
func5();
func6();
...}

Linux Time-Slice Scheduler

- Processes are time-sliced with more time given to lower niceness
- Linux dynamically modifies processes’ time slice according to process behavior
- Processes which block are rewarded with greater percentage of time slice total
Scheduling Methodologies

Time-Slicing with Blocking
- Scheduler shares processor run time between all threads with greater time for higher priority
  - No threads completely starve
  - Corrects for non-"good citizen" threads
  - Can’t guarantee processor cycles even to highest priority threads.
  - More context switching overhead

Thread Blocking Only
- Lower priority threads won’t run unless higher priority threads block (i.e. pause)
  - Requires “good citizen" threads
  - Low priority threads may starve
  - Lower priority threads never break high priority threads
  - Lower context-switch overhead

Linux Default

Notes:
- Linux threads provide extensions for real-time thread behavior as well; however, time-slicing is the default
- Similarly, you can setup BIOS to time-slice threads (TSK’s), but this is not the default for BIOS (i.e. real-time) systems

The Usefulness of Processes

Option 1: Audio and Video in a single Process
```c
// audio_video.c
// handles audio and video in
// a single application

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

Option 2: Audio and Video in separate Processes
```c
// audio.c, handles audio only
int main(int argc, char *argv[])
{
    while(condition == TRUE)
    {
        callAudioFxn();
    }
}
```
```c
// video.c, handles video only
int main(int argc, char *argv[])
{
    while(condition == TRUE)
    {
        callVideoFxn();
    }
}
```

Splitting into two processes is helpful if:
1. audio and video occur at different rates
2. audio and video should be prioritized differently
3. multiple channels of audio or video might be required (modularity)
4. memory protection between audio and video is desired
**Terminal Commands for Processes**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># ps</code></td>
<td>Lists currently running user processes</td>
</tr>
<tr>
<td><code># ps -e</code></td>
<td>Lists all processes</td>
</tr>
<tr>
<td><code># top</code></td>
<td>Ranks processes in order of CPU usage</td>
</tr>
<tr>
<td><code># kill &lt;pid&gt;</code></td>
<td>Ends a running process</td>
</tr>
<tr>
<td><code># renice +5 -p &lt;pid&gt;</code></td>
<td>Changes time-slice ranking of a process (range +/- 20)</td>
</tr>
</tbody>
</table>

**Launching a Process – Terminal**

```
user:/workdir/boottcamp/lab_soaln/lab_soaln - Shell - Konsole

Session Edit \Dev Bookmarks Settings Help

root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release# /lab6_soaln &
[1] 973
root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release# /dev/tty0 initialized with resolution 720x480 and 16 bpp.

root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release# ps

PID TTY TIME CMD
975 pts/0 00:00:00 bash
979 pts/0 00:00:00 lab6_soaln
981 pts/0 00:00:00 ps

root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release# kill 975
root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release# ps

PID TTY TIME CMD
979 pts/0 00:00:00 bash
981 pts/0 00:00:00 ps

[1]+ Terminated /lab6_soaln

root@32.166.10.20:/mnt/boottcamp/lab_soaln/lab_soaln/release#  ```
Side Topic – Creating New Processes in C

We won’t actually need this for our lab exercises, though, we found it interesting enough to include it here.

- fork
- exec
- fork + exec

*Splits one executing process into two with same environment*

*New process replaces old but maintains previous environment*

*Launch new process and keep previous process*

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

Processes and Threads

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

Threads vs Processes

<table>
<thead>
<tr>
<th></th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory protection</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Ease of use</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Start-up cycles</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Context switch</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Codec Engine can span</td>
<td>✗ *Note</td>
<td>✓</td>
</tr>
<tr>
<td>Shared globals</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Environment</td>
<td>program</td>
<td>function</td>
</tr>
</tbody>
</table>

*Note: New Codec Engine feature now supports CE across processes when using LAD (Link Arbitor Daemon).
Entry point at main() for each process is scheduled as a thread.

Threads are scheduled with time slicing and blocking as previously discussed for processes.

Processes may then add additional threads to be scheduled.

---

**Thread Functions**

Threads can only be launched from within a C program (not from a shell)

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, 
   void *(*start_routine)(void *), void *arg);

pthread_join(pthread_t thread, void **retval);

pthread_exit(void *retval);
```
pThread Functions – Create & Exit

- Use `pthread_create()` to kickoff a new thread (i.e. child)
  - Starts new thread executing in the same process as its parent
  - As shown, both threads now compete for time from the Linux scheduler
  - Two important arguments – thread object, function to start running upon creation

- `pthread_exit()` causes child thread end
  - If `_create`'s starting function exits, `pthread_exit()` is called implicitly

Waiting for the Child to Exit

- `pthread_create(&video_thread, null, video_thread_fxn, ...)`
- `pthread_exit(NULL)`
- `main` block waiting for new thread to exit
- `new` thread blocked
- `pthread_create(&new,...)`
Re-Joining Main

```
main

pthread_create(&new,…)

new

main

pthread_create(&new,…)

BlockWaitingForNewThreadToExit(&new)
pthread_join(&new)
```

Multiple Threads ... With or Without Join

Two pThreads without Join

```
main()  

pthread_create(&vid,...)  

video thread

audio thread

pthread_create(&aud,...)
```

Two pThreads with Join

```
main()  

pthread_create(&vid,...)  

video thread

audio thread

pthread_create(&aud,...)  

pthread_join(&vid,...)  

pthread_join(&aud,...)
```

faster
Thread Synchronization

Thread Synchronization (Polling)

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

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

---

### Polling Loop

Start polling for `devComplete`

```
thread A  (higher priority)

process data
```

Process data

```
thread B
```

Wasted cycles while thread A does nothing!
(and thread B starves)
Thread Synchronization (Blocking)

void *threadA(void *env){
while(1){
    read(env->audioFd, env->bufferPtr, env->bufsize); // Blocks (waits till complete)
    doSomethingNext(env->bufferPtr);
}

- Instead of polling on a flag, the thread blocks execution as a result of the driver's read call

- More efficient than polling because thread A doesn’t waste cycles waiting on the driver to fill the buffer

Thread Synchronization (Blocking)

- Semaphores are used to block a thread’s execution until occurrence of an event or freeing of a resource
- Much more efficient system
### Synchronization with Peripherals

<table>
<thead>
<tr>
<th>Driver</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSS driver</td>
<td>read function is blocking</td>
</tr>
<tr>
<td></td>
<td>write function blocks if outgoing buffer full</td>
</tr>
<tr>
<td>V4L2 driver</td>
<td>VIDIOC_DQBUF ioctl is blocking</td>
</tr>
<tr>
<td>FBDEV driver</td>
<td>FBIO_WAITFORVSYNC ioctl is blocking</td>
</tr>
</tbody>
</table>
Using Real-Time Threads

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

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

Time-Sliced A/V Application Analysis

- Audio thread completes 80% of samples
- Video thread drops 6 of 30 frames
- User response delayed 1mS

All threads suffer, but not equally:
- Audio thread real-time failure is highly perceptible
- Video thread failure is slightly perceptible
- Control thread failure is not remotely perceptible

Note:
Time-slicing may also cause real-time failure in systems that are <100% loaded due to increased thread latency
In Linux, Real-Time threads are scheduled according to priority (levels 1-99, where time-slicing is effectively level 0).

The highest priority thread always “wins” and will run 100% of the time unless it blocks.

Audio thread is guaranteed the bandwidth it needs.

Video thread takes the rest.

Control thread never runs!
**Time-Sliced A/V Application Analysis**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio Thread</td>
<td>Audio thread completes, no distortion</td>
</tr>
<tr>
<td>Video Thread</td>
<td>Video thread drops 1 of 30 frames</td>
</tr>
<tr>
<td>Control Thread</td>
<td>No user response</td>
</tr>
</tbody>
</table>

**Still a problem:**
- Audio thread completes as desired
- Video thread failure is practically imperceptible
- Control thread never runs – User input is locked out

**Hybrid A/V Application, >100% load**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio Thread</td>
<td>Audio thread is guaranteed the bandwidth it needs</td>
</tr>
<tr>
<td>Video Thread</td>
<td>Video thread takes <em>most</em> of remaining bandwidth</td>
</tr>
<tr>
<td>Control Thread</td>
<td>Control thread gets a small portion of remaining bandwidth</td>
</tr>
</tbody>
</table>

Audio Thread: (priority 99)  
Video Thread: (pri 0, nice -5)  
Control Thread: (pri 0, nice +5)
Hybrid A/V Application Analysis

<table>
<thead>
<tr>
<th>Thread Type</th>
<th>Status</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio Thread</td>
<td>Audio thread completes, no distortion</td>
<td></td>
</tr>
<tr>
<td>Video Thread</td>
<td>Video thread drops 2 of 30 frames</td>
<td></td>
</tr>
<tr>
<td>Control Thread</td>
<td>User response delayed 100ms</td>
<td></td>
</tr>
</tbody>
</table>

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

Default Thread Scheduling

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

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

<table>
<thead>
<tr>
<th>pThread attributes:</th>
<th>NULL / default value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>stacksize</td>
<td>PTHREAD_STACK_MIN</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>detachedstate</td>
<td>PTHREAD_CREATE_JOINABLE</td>
</tr>
<tr>
<td>schedpolicy</td>
<td>SCHED_OTHER (time slicing)</td>
</tr>
<tr>
<td>inheritsched</td>
<td>PTHREAD_INHERIT_SCHED</td>
</tr>
<tr>
<td>schedparam.sched_priority</td>
<td>0</td>
</tr>
</tbody>
</table>
Scheduling Policy Options

<table>
<thead>
<tr>
<th>Sched Method</th>
<th>SCHED_OTHER</th>
<th>SCHED_RR</th>
<th>SCHED_FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT priority</td>
<td>Time Slicing</td>
<td>Real-Time (RT)</td>
<td></td>
</tr>
<tr>
<td>Min niceness</td>
<td>+20</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Max niceness</td>
<td>-20</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Scope</td>
<td>root or user</td>
<td>root</td>
<td>root</td>
</tr>
</tbody>
</table>

- Time Sliced scheduling is specified with SCHED_OTHER:
  - Niceness determines how much time slice a thread receives, where higher niceness value means less time slice
  - Threads that block frequently are rewarded by Linux with lower niceness

- Real-time threads use preemptive (i.e. priority-based) scheduling
  - Higher priority threads always preempt lower priority threads
  - RT threads scheduled at the same priority are defined by their policy:
    - SCHED_FIFO: When it begins running, it will continue until it blocks
    - SCHED_RR: "Round-Robin" will share with other threads at it's priority based on a deterministic time quantum

Real-time Thread Creation Procedure

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

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

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

// Set the scheduler priority via audioThreadParams struct
audioThreadParams.sched_priority = 99;

// Create the new thread using thread attributes
pthread_create(&audioThread, &audioThreadAttrs,
               audio_thread_fxn, (void *) &audio_env);
```
Lab 8

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.

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 RedHat 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.x470MV application to app_AUDIO.x470MV.
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.x470MV application using the following command:
   
   ./app_AUDIO.x470MV &

   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.)

7. On the DVEVM board, execute the app_DEBUG.x470MV application (the video loopthru application) using the following command:
   
   ./app_DEBUG.x470MV

   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.x470MV"

   One more thing to try:
   
   ps -e

   . What does this do ?

10. Halt the audio loopthru using the kill command and PID value from the last step.

    kill -s SIGINT <app_AUDIO.x470MV process ID>
Lab08a Question

1. What 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/workshop directory.
   
   cd ~/workshop

   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. 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.

13. 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

14. Open lab08b_audio_video/app/main.c in a text editor.

15. Fill in the missing .h file, as well as the missing _env variable to main.c.

Your main.c file should contain the following:

```c
#include "video_thread.h"
#include "video_env"  // (which is the video_thread_env variable)

#include "audio_thread.h"
#include "audio_env"  // (which is the audio_thread_env variable)
```

You need to add the following to main.c. (Refer to lab06c for this code.)

```c
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting..\n");

    __________________________
    video_env.quit = 1;
}
```

16. Make sure both video and audio 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 a similar statement for the audio_env.quit to the signal handler below.

```c
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting..\n");

    __________________________
    video_env.quit = 1;
}
```

17. Include the header file that is needed to call pthread functions.

This one is kind of obvious, but the header file name is: pthread.h

18. Declare two pthread handles and two return pointers needed to manage our two new threads (audio, video).

At the top of main(), add two pthread handles (of type pthread_t) named audioThread and videoThread. Handles are used to refer to instantiated objects; the handle value will be set during pthread_create() in step 19, then used again later to refer to that pthread instance.

Also, we want to add two void pointers: one named audioThreadReturn, the other named videoThreadReturn. The return pointers will be used when “joining” (i.e. exiting) a thread in step 21; they will allow you to interrogate the status after an exit.

It should end up looking like:

```c
pthread_t    audioThread, videoThread;
void         *audioThreadReturn;
void         *videoThreadReturn;
```
19. Replace a direct function call with the creation of a video (and audio) pThread.

Currently `main.c` calls `video_thread_fxn`. Replace this direct function call with two calls to `pthread_create()`, one for the `video_thread_fxn` and the second for `audio_thread_fxn`.

Recall from our discussion:

```c
/* Call video thread function */
videoThreadReturn = video_thread_fxn((void *) &video_env);
```

Replace the following:

```c
/* Create a thread for video loopthru */
if(pthread_create(&thread, NULL, video_function, argument) != 0){
    ERR("pthread create failed for video thread\n");
    status = EXIT_FAILURE;
    goto cleanup;
}
initMask |= VIDEOTHREADCREATED;
```

For additional information on `pthread()`, you can check out the optional sidebar discussion at the end of the chapter (page 8-29).
20. 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.

```c
#ifdef _DEBUG_
    sleep(1);
#endif
```

Don’t forget to include the proper header file for the sleep() function: unistd.h

21. “cleanup” using pthread_join() for both audio and video threads.

First, let’s create a “cleanup” section in our main.c file.

After both the audio and video threads have been created, use pthread_join on both threads to pause execution of the main thread until both threads have exited.

The prototype for pthread_join is:

```c
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 18, then filled-in with pthread_create() in step 19).

We use audioThreadReturn 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:

```c
(void **) &videoThreadReturn
(void **) &audioThreadReturn
```

22. Build and install the application using gMake:

```bash
make debug install
```

23. Execute the ./app_DEBUG.x470MV application on the DVEVM board.

You should have audio and video playing through the board simultaneously.

24. Press Ctrl-C to exit the application.

Lab08b Question

- What scheduling policy is being used by each of the audio and video threads? ___________
- ________________________________________________________________________________
- ________________________________________________________________________________
- ________________________________________________________________________________
Lab 08c_audio_video_rtime

This lab extends lab08b_audio_video by utilizing the pthread_attr structure to specify real-time behavior for the threads created.

Most of what needs to be added are tedious, rote settings of various parameters; therefore, we will treat this as an inspection-only lab.

Examine Files

25. Copy your osdfiles (especially osd.r16) from lab08b to lab08c.

26. Change to the /home/user/workshop/lab08c_audio_video_rtime/app directory.

27. Examine main.c.
   You will see that instead of passing a NULL value to the attribute pointer of the pthread_create() function, this version of the application declares audioThreadAttrs and videoThreadAttrs pthread attributes structures and then initializes them using the pthread_attr_init method. Various methods are used to set different parameters within these thread attributes structures, including pthread_attr_setschedpolicy method to set the scheduling policy to SCHED_RR, the round-robin real-time scheduling policy.

Build, Install, Run Application

28. Build and install the application using gMake.

29. Execute the ./app_DEBUG.x470MV application on the DVEVM board.
   You should notice no perceptible difference between the execution of this application versus the one in lab08b_audio_video. Both applications should loop through audio and video simultaneously with no distortion.

30. Press Ctrl-C to exit the application.

Questions

1. In lab08c_audio_video_rtime/main.c, what priority is the audio thread set to? (You may use an expression here.)

2. When running the debug version of the application (../app_DEBUG.x470MV), what does the debug output indicate the audio thread priority is set to? (Numerical value)
3. What priority is the video thread set to? (You may use an expression here.)

4. When running the debug version of the application (./app_DEBUG.x470MV), what does the debug output indicate the video thread priority is set to? (Numerical value)

5. Why don’t our audio and video threads end with pthread_exit()? How can our system work without these functions?

Lab08d_challenge1 -- Control Thread

31. Begin by copying the working lab08c_audio_video_rtime into lab08c_challenge1

   cp -R -f lab08c_audio_video_rtime/* lab08d_challenge1

32. Modify lab08c_challenge1/app/main.c to add a control thread that scans an input string using scanf, and, depending on the command, either creates a new audio or video thread or deletes the currently running audio or video thread using pthread_join.

   (You should also keep track of the state of the audio and video thread and not allow creation of more than one audio or video thread or allow deletion when there is no audio or video thread.)

   Your control thread should recognize and respond to the following control commands:

   create_video
delete_video
create_audio
delete_audio
quit
Sidebar - Looking at the pthread arguments in detail:

The detailed function prototype for pthread_create is:

```c
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_type`).

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_type`.

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.:

```c
(void *) &audio_env
(void *) &video_env
```
Sidebar to the Sidebar – The devilish details of the \texttt{pthread\_create() 3\textsuperscript{rd} arg}

The third argument to the \texttt{pthread\_create()} function specifies the \texttt{start\_function}. That is, the function automatically run after creating the new thread.

As an example, the argument might look like:

\begin{verbatim}
pthread_create(&myThread, NULL, \textbf{myFunction},(void *) &myArg)
\end{verbatim}

Looking at the \textit{official} definition for the \texttt{pthread\_create()} function, we find the third argument looks like:

\begin{verbatim}
void *(\*start\_routine)(void *)
\end{verbatim}

All this is really saying is that this argument is just a pointer to a function whose prototype is:

\begin{verbatim}
void *start\_routine(void *arg);
\end{verbatim}

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 \texttt{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 \texttt{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 \texttt{video\_thread\_fxn} and \texttt{audio\_thread\_fxn} that we have been using happen to both use void pointers as their argument and return values.
Introduction

In this chapter the steps required to use a given engine will be examined

Learning Goals

In this chapter the following topics will be presented:

- VISA (Class) API
- Under the Hood of the Codec Engine
- Detailed Look at VISA Functions
- Using the Codec Engine within Multiple Threads
Chapter Topics

Local Codecs - Using a Given Engine ................................................................. 9-1

VISA (Class) API .................................................................................................. 9-3

Under the Hood of the Codec Engine ............................................................... 9-7

Detailed Look at VISA Functions ........................................................................ 9-8

Rules for Opening and Using Engines ............................................................... 9-14

Lab 9 ..................................................................................................................... 9-15
  Labs 9-12 Uses Cases ....................................................................................... 9-16
  Lab09a_use_published_engine ......................................................................... 9-17
  Lab09b_use_published_engine_av ................................................................. 9-21
  Lab09c_challenge1 — Control Thread .............................................................. 9-29

DMAI (Digital Media App Interface) Library ..................................................... 9-30

Codec Engine Functions Summary ................................................................. 9-32
VISA (Class) API

Calling all Codecs

The Application has access to both Local (ARM-side) xDM codecs and Remote (DSP-side) xDM codecs.

Calling all Codecs

The Application has access to both Local (ARM-side) xDM codecs and Remote (DSP-side) xDM codecs.
The Application has access to both Local (ARM-side) xDM codecs and Remote (DSP-side) xDM codecs.
The Application has access to both Local (ARM-side) xDM codecs and Remote (DSP-side) xDM codecs.

Conceptual View – Local Engine

The Application Interfaces to the Codec Engine Framework Through:

**Engine Functions**
- CERuntime_init
- Engine_open
- Engine_close
- CERuntime_exit() (CE 1.20+)

**Class (VISA) Functions**
- VIDENC_create
- VIDENC_control
- VIDENC_process
- VIDENC_delete
Master Thread Key Activities

```c
idevfd = open("/dev/xxx", O_RDONLY);
ofilefd = open("./fname", O_WRONLY);
oct(idevfd, CMD, &args);
CERuntime_init();
myCE = Engine_open("vcr", myCEAttrs);
myVE = VIDENC_create(myCE, "videnc", params);
while( doRecordVideo == 1 ) {
    read(idevfd, &rd, sizeof(rd));
    VIDENC_control(myVE, ...);
    VIDENC_process(myVE, ...);
    write(ofilefd, &wd, sizeof(wd));
}
close(idevfd);
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

Note: the above pseudo-code does not show double buffering, often essential in R/T systems!

Codec Engine - Use Cases

Lab 9

Given

Lab 10

Engine

Lab 11

Engine

Lab 12

DSP Server

Lab 9

Given

Lab 11

Given

Lab 12

DSP Server

Note: the above pseudo-code does not show double buffering, often essential in R/T systems!
Create and Delete

- The application creates a local (or remote) video encoder instance through the VIDENC_create API
- The VIDENC_create or VIDENC_delete function passes the request to the Engine, which
  - determines if the requested codec is local via the codec table
  - And, if the codec is local, grants or frees resources such as memory and DMA channels to/from the algorithm
  - These resources ultimately come from the Linux O/S, which the Engine accesses via its O/S Abstraction Layer

Control and Process

- The application accesses a codec instance through VIDENC_control and VIDENC_process API
- The VIDENC_control and VIDENC_process functions call corresponding control or process function from the Codec.
- Control and process calls made via a function pointer in the VIDENC_object
- Reason for this extra mechanism will become more clear when we study remote codecs
Detailed Look at VISA Functions

Calling \texttt{Engine\_open()} and \texttt{\_create()} (1)

\begin{verbatim}
Engine\_Handle myCE;
AUDENC\_Handle myAE;
AUDENC\_Params params;

CERuntime\_init();
myCE = Engine\_open(\textasciitilde{\texttt{myEngine}}, NULL);

... setup params structure

myAE = AUDENC\_create(myCE, \textasciitilde{\texttt{myEncl}}, &params);
\end{verbatim}

- Engine and Codec names are declared during the \texttt{engine\_config} step of the build process
- We will explore this in the next chapter

\textbf{xDM : Creation Parameters}

\begin{verbatim}
typedef struct IAUDDEC\_Params {
    XDAS\_Int32 size;
    XDAS\_Int32 maxSampleRate;
    XDAS\_Int32 maxBitrate;
    XDAS\_Int32 maxNoOfCh;
    XDAS\_Int32 dataEndianness;
} IAUDDEC\_Params;
\end{verbatim}

\begin{verbatim}
typedef struct IVIDDEC\_Params {
    XDAS\_Int32 size;
    XDAS\_Int32 maxHeight;
    XDAS\_Int32 maxWidth;
    XDAS\_Int32 maxFrameRate
    XDAS\_Int32 maxBitRate;
    XDAS\_Int32 dataEndianness;
    XDAS\_Int32 forceChromaFormat;
} IVIDDEC\_Params;
\end{verbatim}
Calling `process()`

```c
XDM_BufDesc inBuf, outBuf;
AUDENC_InArgs encInArgs;
AUDENC_OutArgs encOutArgs;
Int status;
encInArgs.size = sizeof(encInArgs);
encOutArgs.size = sizeof(encOutArgs);

... fill in remaining encInArgs values
... setup the Buffer Descriptors

status = AUDENC_process(myAE, &inBuf, &OutBuf,
                            &encInArgs, &encOutArgs);

if(status != 0) doerror(status);
```

Audio Decoder Process Arguments

```c
typedef struct XDM_BufDesc {
    XDAS_Int32 numBufs;
    XDAS_Int32 *bufSizes;
    XDAS_Int8 **bufs;
} XDM_BufDesc;

typedef struct IAUDDEC_InArgs {
    XDAS_Int32 size;
    XDAS_Int32 numBytes;
} IAUDDEC_InArgs;

typedef struct IAUDDEC_OutArgs {
    XDAS_Int32 size;
    XDAS_Int32 extendedError;
    XDAS_Int32 bytesConsumed;
} IAUDDEC_OutArgs;
```

// for buffer description (input and output buffers)
// number of buffers
// array of sizes of each buffer in 8-bit bytes
// pointer to vector containing buffer addresses

// for passing the input parameters for every decoder call
// size of this structure
// size of input data (in bytes) to be processed

// relays output status of the decoder after decoding
// size of this structure
// Extended Error code. (see XDM_ErrorBit)
// Number of bytes consumed during process call
Video Decoder Process Arguments

typedef struct XDM_BufDesc {
  XDAS_Int32 numBufs; // number of buffers
  XDAS_Int32 *bufSizes; // array of sizes of each buffer in 8-bit bytes
  XDAS_Int8 **bufs; // pointer to vector containing buffer addresses
} XDM_BufDesc;

typedef struct IVIDDEC_InArgs {
  XDAS_Int32 size; // size of this structure
  XDAS_Int32 numBytes; // Size of valid input data in bytes in input buffer
  XDAS_Int32 inputID; // algo tags out frames with this (app provided) ID
} IVIDDEC_InArgs;

typedef struct IVIDDEC_OutArgs {
  XDAS_Int32 size; // size of this structure
  XDAS_Int32 extendedError; // extended error code
  XDAS_Int32 bytesConsumed; // bytes consumed per given call
  XDAS_Int32 decodedFrameType; // frame type
  XDAS_Int32 outputID; // output ID: tagged w value from *InArgs:InputId
  IVIDEO_BufDesc displayBufs; // buffer pointers for current displayable frames.
} IVIDDEC_OutArgs;

Buffer Descriptors

AUDENC_process(myAE, &ibuf, &obuf, &myInArgs, &myRetVals);

- xDM Buffer Descriptors are used to pass buffers to and from the _process() function
- Provides a standardized way for passing any number of arbitrarily sized buffers
### Example Buffer Descriptor

**Allocate Buffers**

- XDAS_Int8 buf0[2];
- XDAS_Int8 buf1[6];
- XDAS_Int8 buf2[4];

**Allocate Buffer Descriptor**

- XDM_BufDesc inBufDesc;
- XDAS_Int32 in BufSizes[3];
- XDAS_Int8 *inBufPtrs[3];

**Build Buffer Descriptor**

- inBufDesc.numBufs = 3;
- inBufDesc.bufSizes = inBufSizes;
- inBufDesc.bufs = inBufPtrs;

**Set size & Pointer Arrays**

- inBufSizes[0] = sizeof(buf0);
- inBufSizes[1] = sizeof(buf1);
- inBufSizes[2] = sizeof(buf2);
- inBufPtrs[0] = buf0;
- inBufPtrs[1] = buf1;
- inBufPtrs[2] = buf2;

---

### Buffer Descriptors

```c
typedef struct XDM_BufDesc {
    XDAS_Int32 numBufs;
    XDAS_Int32 *bufSizes;
    XDAS_Int8 **bufs;
} XDM_BufDesc;

typedef struct XDM1_SingleBufDesc {
    XDAS_Int8 *buf;
    XDAS_Int32 bufSize;
    XDAS_Int32 accessMask;
} XDM1_SingleBufDesc;

typedef struct XDM1_BufDesc {
    XDAS_Int32 numBufs;
    XDM1_SingleBufDesc descs[];
} XDM1_BufDesc;
```

* XDAM Buffer Descriptors are used to pass buffers to and from the _process() function

```c
#define XDM_ACCESSMODE_READ 0
#define XDM_ACCESSMODE_WRITE 1
```

```c
#define XDM_ISACCESSMODE_READ(x)
#define XDM_ISACCESSMODE_WRITE(x)
#define XDM_SETACCESSMODE_READ(x)
#define XDM_SETACCESSMODE_WRITE(x)
#define XDM_CLEARACCESSMODE_READ(x)
#define XDM_CLEARACCESSMODE_WRITE(x)
```
### Calling _control() and _delete()

```c
AUDENC_Status status;
AUDENC_DynParams dynParams;
Int retVal;

retVal = AUDENC_control(myAE, XDM_GETSTATUS,
                        &dynParams, &status);

if(retVal !=0) printf("AUDENC_control Returned
                   extended error %d\n", status.extendedError);
```

### Calling AUDENC_delete()

```c
AUDENC_delete(myAE);
```

### xDM control() API

```c
Int (*control) (IAUDDEC_Handle handle, IAUDDEC.Cmd id,
                IAUDDEC_DynamicParams *params, IAUDDEC_Status *status)
```

- **handle**: pointer to instance of the algorithm
- **cmdId**: for controlling operation of the control
  - **XDM_GETSTATUS**: returns status of the last decode call in IAUDDEC_Status structure
  - **XDM_SETPARAMS**: initializes decoder via IAUDDEC_DynamicParams structure
  - **XDM_RESET**: resets the decoder
  - **XDM_SETDEFAULT**: sets decoder parameters to default set of values
  - **XDM_FLUSH**: the next process call after this control command will flush the outputs
  - **XDM_GETBUFINFO**: provides input and output buffer sizes
- **params**: structure that allows the parameters to change on the fly of the process call
- **status**: status of decoder as of the last decode call is written to IAUDDEC_Status structure

```c
typedef struct IAUDDEC_DynamicParams {
    // control API argument
    XDAS_Int32 size;
    // size of this structure
    XDAS_Int32 outputFormat;
    // sets interleaved/Block format. see IAUDIO_PcmFormat
} IAUDDEC_DynamicParams;
```
Audio Decoder Control Arguments

typedef struct IAUDDEC_Status {
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 extendedError; // extended error code. (see XDM_ErrorBit)
    XDAS_Int32 bitRate; // Average bit rate in bits per second
    XDAS_Int32 sampleRate; // sampling frequency (in Hz)
    XDAS_Int32 numChannels; // number of Channels: IAUDIO_ChannelId
    XDAS_Int32 numLFEChannels; // number of LFE channels in the stream
    XDAS_Int32 outputFormat; // output PCM format: IAUDIO_PcmFormat
    XDAS_Int32 autoPosition; // support for random position decoding: 1=yes 0=no
    XDAS_Int32 fastFwdLen; // recommended FF length in case random access in bytes
    XDAS_Int32 frameLen; // frame length in number of samples
    XDAS_Int32 outputBitsPerSample; // no. bits per output sample, eg: 16 bits per PCM sample
    XDM_AlgBufInfo bufInfo; // input & output buffer information
} IAUDDEC_Status;

typedef struct XDM_AlgBufInfo {
    XDAS_Int32 minNumInBufs; // min number of input buffers
    XDAS_Int32 minNumOutBufs; // min number of output buffers
    XDAS_Int32 minInBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req'd for each input buffer
    XDAS_Int32 minOutBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req'd for ea. output buffer
} XDM_AlgBufInfo;

Video Decoder Control Arguments

typedef struct IVIDDEC_Status {
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 extendedError; // Extended Error code. (see XDM_ErrorBit)
    XDAS_Int32 outputHeight; // Output Height
    XDAS_Int32 outputWidth; // Output Width
    XDAS_Int32 frameRate; // Average frame rate* 1000
    XDAS_Int32 bitRate; // Average Bit rate in bits/second
    XDAS_Int32 contentType; // IVIDEO_PROGRESSIVE or IVIDEO_INTERLACED
    XDM_AlgBufInfo bufInfo; // Chroma output fmt of type IVIDEO_CHROMAFORMAT
} IVIDDEC_Status;

typedef struct XDM_AlgBufInfo {
    XDAS_Int32 minNumInBufs; // min number of input buffers
    XDAS_Int32 minNumOutBufs; // min number of output buffers
    XDAS_Int32 minInBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req'd for each input buffer
    XDAS_Int32 minOutBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req'd for ea. output buffer
} XDM_AlgBufInfo;
Rules for Opening and Using Engines

Engine Rules

- Only one Engine can be open at a time*
- The Engine can: Only be accessed from within a single process *be accessed across multiple threads within a single process
- All threads must open the same Engine (i.e. Engine's name must match)
- Each thread should obtain its own Engine handle
  - Often, main thread creates environment variable – with engine "name" – to pass to each thread which uses CE

```c
void *myVideoThread(void *env) {
  Engine_Handle hvEngine;
  VIDENC_Handle hVidenc;
  hvEngine = Engine_open(env->eName);
  hVidenc = VIDENC_create(hvEngine,…);
  VIDENC_process(hVidenc,…);
}
```

```c
void *myAudioThread(void *env) {
  Engine_Handle haEngine;
  AUDENC_Handle hAudenc;
  haEngine = Engine_open(env->eName);
  hAudenc = AUDENC_create(haEngine,…);
  AUDENC_process(hAudenc,…);
}
```

*Notes: Codec Engine 2.0 supports use across processes when using LAD (Link Arbitor Daemon).
See http://tiexpressdsp.com/index.php/Multiple_Threads_using_Codec_Engine_Handle

The first rule is that only one Engine can be open at a time. While not strictly accurate, it is a good rule to live by. Technically speaking, while more than one Engine can be opened at a time, only one DSP Server can be opened – and thus loaded into the DSP – at a time. This means that if you try to open multiple Engines, they must either contain only local codecs, or they must reference the same DSP Server. The bottom line is that there is rarely – if ever – a need to have more than one Engine open at a time.

An Engine can only be accessed from within a single process. It is very common, though that an Engine may be accessed by many threads (i.e. pThreads) as needed within that process.

If you use an Engine across multiple threads, each thread should call Engine_open(). In our previous example, we have passed the Engine name to each thread in a variable called *env. Calling Engine_open() multiple times does not actually re-open the same Engine, rather, it allows the Codec Engine framework to count the number of threads using the Engine. Thus, if later on one thread calls Engine_close(), it won’t actually close the Engine until it has been closed the by all the threads who earlier called _open().
Lab 9

In this lab, you will extend the lab8 audio/video example to add a video encoder and decoder into the video loopthru thread (lab 9a) and an audio encoder and decoder into the audio thread (lab 9b). For this lab we will be using dummy (i.e. pass-thru) audio and video codecs that come with the Codec Engine examples, running locally on the ARM processor.

The project uses a pre-built or “published” engine; essentially, all of the signal processing content has been wrapped up for you in 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. (Later on, our exercises will examine building engines.)
Labs 9-12 Uses Cases

Codec Engine - Use Cases

- There are four different use-cases for interacting with the Codec Engine
- Here's a quick description, along with a reference where it's covered

Lab 9
- Linux application dev'l only – Engine provided as .o lib
- Engine provided by another in-house team or 3rd Party
- Modularity allows to focus on end-app
- Codecs local or remote (don’t care)

Lab 10
- Linux application dev'l – as well as Engine config (.cfg)
- Build Engine & app all-in-one
- While easy, user must know about pkgs & .cfg files
- Local codecs only

Lab 11
- Similar to prev use-case, but you're provided server containing remote codecs
- Simple (.cfg) method makes remote codecs easy to use

Lab 12
- Similar to others from app/engine producers persp.
- Building a server is like building a DSP executable (i.e. “.out” file)
- Requires some embedded syst. dev'l experience

Lab 9
Lab09a_use_published_engine

To reconfirm, in this lab you should see video and hear audio, but we are only “processing” the video. That is, we will continue to use a **memcpy()** in `main.c` to pass-thru the audio. The video will go thru our copy-based video encoder and decoder. (In lab09b we will add the code required to get the audio copy codecs running, as well.)

**Examine Files.**

1. **Change to the** `/home/user/workshop/lab09a_use_published_engine` **directory and list the contents.**

   Note that there are now two directories: **app** and **engine**. The **engine** directory contains two object libraries named: `engine_debug.o` and `engine_release.o`. These object files contain all object code needed to add the given engine into an application. Hence these are published as **pre-built** engines. In lab10 we will examine how to build or modify an engine.

2. **Change to the** `app` **directory and examine the files:**

   `video_decoder.c`

   The `video_decoder.c` file defines helper functions for setting up (i.e. creating), using and cleaning up (deleting) a video decoder using the codec engine. The corresponding functions are **setup_video_decoder**, **decode_video** and **cleanup_video_decoder**. These functions are utilized within `video_thread.c`.

   `video_encoder.c`

   Similar to `video_decoder.c`, but for an encoder.

   `video_thread.c`

   `video_decoder_setup` and `video_encoder_setup` (as defined in `video_decoder.c` and `video_encoder.c`) are added to the setup phase of the thread to create and configure a video encoder and decoder from the codec engine. During the **while** loop of the thread, `encode_video` and `decode_video` calls are added into the loopthu to first encode the incoming video frame and then decode again before sending to the video output driver. Finally, after the loop exits and the thread enters the cleanup phase, `video_encoder_cleanup` and `video_decoder_cleanup` are used to free these resources.

**Examine main.c**

Note that **CERuntime_init()** must be called before any other Codec Engine functions. Also note that the `video_env` structure has been modified to include a string value of `engineName` so that main can pass the name of the Codec Engine down to the video 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).

4. Insert the CMEM driver into the kernel.

   Remember, in order to use a driver it must be installed into the kernel. MontaVista does not include either the CMEM (contiguous memory allocator) or the DSPLINK (DSP link interprocessor communications) drivers in the default build of the Linux Kernel; rather, they are provided as two kernel modules which will need to be inserted before using them.

   Starting with Lab 9, we’ll 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 `insmod` command and allocates the appropriate device nodes to support the drivers using `mknod`, as we discussed back in Chapter 6.

   Follow the following two steps to get these drivers working:

   • Verify that following files are in your `/opt/workshop` directory on the DVEVM. If they’re not present, install them by running the `install.sh` script from the
     `~/workshop/lab00a_install_scripts` directory.

     `cmem.ko`
     `dsplink.ko`
     `loadmodules.sh`
     `unloadmodules.sh`

   • On the DVEVM board, run the script:

     `./loadmodules.sh`

     If the loadmodules script fails, power cycle the DVEVM and try again. If that does not work, it is probable that someone has updated (or modified) the Linux kernel. Ask your instructor for help if you have questions and need help with this.

5. Execute the app_DEBUG.x470MV application.

   After you’ve confirmed it works, press Ctrl-C to exit the application.
Questions

Shown below is an “infomap” of how the files and functions within the files are connected. Sometimes it is difficult to see the forest because you’re IN the forest. So, the slide below will help (to some degree) show what is going on graphically. This will also help your understanding of the files and what they do for the future labs 9b-12. Feel free to refer to this slide as often as you like.

Now that you’ve taken a small amount of time to digest the above slide, let’s focus on where the actual VISA functions are used. The whole point of this chapter is to understand the VISA function calls, so this is your chance to peruse the files and examine how these calls are made by answering the following questions:

1. Which file and function contains the call to **CERuntime_init()**?
   File: ____________   Function: __________

2. Which function contains the call to **Engine_open()**? We’ll give you the file name…
   File: **engine.c**   Function: ______________________

3. Which file/function contains the call to **Engine_close()**? Hint: it is in the same file as **Engine_open()**.
   File: ____________   Function: ______________________
4. Now, we need to find the actual VISA functions (create, process, control, delete) in the source files. Which source file would you THINK would contain the VISA functions for ENCODE?
   File: ________________________

5. Ok, now that you’ve had a shot at what you THINK, let’s find out if you were right. The VISA functions for encode and decode are in their corresponding `video_encoder.c` and `video_decoder.c` files. Let’s just look at ENCODE for now (decode is similar). Open up `video_encoder.c`. Which function contains the call to `VIDENC_create()`?
   Function that contains call to `VIDENC_create`: ____________________________

6. What three parameters are sent to `VIDENC_create()`?
   `VIDENC_create ( ________________, ________________, ______ );`

7. Function that contains call to `VIDENC_process`: ____________________________

8. How many parameters are needed for the call to `VIDENC_process`?
   _____

9. Move down in the `video_encoder.c` file to find the function `video_encoder_cleanup()`. Locate the call to `VIDENC_control()`. How many parameters are needed for the call to `VIDENC_control`?
   _____

10. Fill in the single parameter used to call `VIDENC_delete()`:
    `VIDENC_delete ( _________________ );`
Lab09b_use_published_engine_av

Using lab09a_use_published_engine as a reference, add an audio encoder and decoder into the audio thread of the application. To the previous lab, you will need to modify audio_thread.c to use:

- audio_encoder_setup()
- encode_audio()
- audio_encoder_cleanup()

As a hint, you may want to use video_thread.c as a reference, even cutting and pasting segments from this file as a starting point for your code.

Remember that our “engine” uses copy-based (dummy) codecs, so it is not necessary to set up static or dynamic parameters. You may use a NULL value for the static parameters (indicates default static parameters) when creating a new encoder or decoder; and you do not need to set dynamic parameters for this lab.

File Management

6. Begin by copying the files of lab09a_use_published_engine into lab09b_use_published_engine_av.
   
   cd ~/workshop
   cp -R -f lab09a_use_published_engine/* lab09b_use_published_engine_av

7. Change to the /home/user/workshop/lab09b_use_published_engine_av/app directory and list the contents.
   Note that you have been provided with four files, audio_encoder.c, audio_encoder.h, audio_decoder.c and audio_decoder.h.

Examine Files

8. Examine the audio_encoder.c and audio_decoder.c, and their associated header files.
   These files are exactly analogous to the corresponding video_encoder and video_decoder files you examined in lab09a_use_published_engine.
Modify audio_thread.c

9. Open lab09b_use_published_engine_av/app/audio_thread.c for editing.

10. Add the appropriate #includes and #defines to allow access to the audio decoder and encoder.

To make this easier, here’s a clip from the video_thread.c we copied from the last lab exercise. In general, it shows us what we need to include to get our new codecs working. You can copy/paste the bolded statements from video_thread.c, but you need to make sure and change the video references to audio.

```c
/* Codec Engine headers */
#include <xdc/std.h>
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/video/viddec.h>
#include <ti/sdo/ce/video/videnc.h>

/* Application header files */
#include "debug.h" // DBG and ERR macros
#include "video_thread.h" // video thread definitions
#include "video_osd.h" // OSD window functions
#include "video_output.h" // Video display device functions
#include "video_input.h" // Video input device functions
#include "engine.h" // Helper utils for engine_open/close
#include "video_decoder.h" // Video decoder functions
#include "video_encoder.h" // Video encoder functions

/* Video encoder and decoder used */
#define VIDEO_ENCODER "video_encoder"
#define VIDEO_DECODER "video_decoder"

/* Intermediate buffer for encoded video */
char *encBuf = NULL; // pointer to encoded buffer
int encBufSize = 0; // size of encoded buffer
int numbytes; // how full a given buffer is
    // that is, numbytes < encBufSize

/* Codec engine variables */
Engine_Handle engineHandle = NULL; // handle to Engine
VIDENC_Handle encoderHandle = NULL; // handle to video encoder
VIDDEC_Handle decoderHandle = NULL; // handle to video decoder
```

Hint: Make sure you change video to audio
10a. First, we need to include the header files that reference the standard XDC definitions, as well as the codec engine and audio codec definitions.

10b. Looking at our video_thread.c, we see that our code calls a couple “wrapper” functions. That is, functions that encapsulate the actual calls to the codec engine. Since these functions are found in three separate files, we need to reference their header files.

10c. As you can see, in our video thread, rather than using the actual codec string names (as defined in our .cfg file), we chose to abstract them via #defines.

You should follow our example by creating two constant defines: AUDIO_DECODER and AUDIO_ENCODER. In this case, though, we’ll leave it up to you to examine the .cfg file published along with the engine to discover the strings they should be set to.

10d. There are a couple of variables/buffers that we need to define. The first part of D creates a buffer for the encoded data, along with its size – and a variable to tell us how full it is, which is returned by the codec (though we cheat in this exercise, as described in step 14.)

The second part of D creates the various handles returned by the engine open and codec create calls.

11. Extend the #define’d bit values for the initMask to include the following.

```c
/* The levels of initialization for initMask */
#define INPUT_OSS_INITIALIZED 0x1
#define INPUT_BUFFER_ALLOCATED 0x2
#define OUTPUT_OSS_INITIALIZED 0x4
#define OUTPUT_BUFFER_ALLOCATED 0x8
#define AUDIOENCODERCREATED 0x10
#define AUDIODECODERCREATED 0x20
#define ENCODEDBUFFERALLOCATED 0x40
#define ENGINEOPENED 0x80
```

Once again, we will use these in our error management code to detect which actions have (or have not) been completed successfully.
Modifying audio_thread.c (cont’d)

12. Open the codec engine and create instances of both the audio encoder and decoder.

In engine.c, audio_encoder.c and audio_decoder.c, we have provided “wrapper” functions setting up these entities. (In fact, we added header files referencing these functions in step 10b.)

In this step you need to add three calls to open the Engine, as well as create instances of both our encoder and decoder codecs. In the following code, we have provided you the first two, letting you work out the third.

As a hint, your code needs to be similar to that found in the video_thread_fxn of video_thread.c shown below; though, you’ll want to replace the “video” references with “audio”. (You can use that file as a copy/paste reference, if you’d like.)

video_thread.c

```c
/* Call setup to open the Engine */
/* Note, open engine from every thread that uses it */
if ( engine_setup ( &engineHandle, envPtr->engineName, NULL) != engine_SUCCESS) {
    ERR("Engine setup failed in video_thread_fxn\n");
    status = VIDEO_THREAD_FAILURE;
    goto cleanup;
}

/* Create an instance of the video encoder */
if ( video_encoder_setup ( engineHandle, VIDEO_ENCODER, &encoderHandle) == VENC_FAILURE) {
    ERR("Video_encoder_setup failed in video_thread_fxn\n");
    status = VIDEO_THREAD_FAILURE;
    goto cleanup;
}
initMask |= VIDEOENCODERCREATED;
```

Similarly, you’ll need to copy the setup code for the Audio Decoder ...

1st Arg: engine handle

2nd Arg: string name of decoder

3rd Arg: address of the decoder’s handle – to be returned by this function

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? ________________________________

________________________________________________________________________

Why do we pass a NULL in our engine_setup() function? _______________________

________________________________________________________________________
Modifying audio_thread.c (cont’d)

13. Allocate buffer to hold data between encoder and decoder process calls in audio_thread.c.

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 item that can be called via their _control() function.

In any case, since we are using dummy codecs (shipped with the Codec Engine), this is not an option. Therefore, let’s just set the encoded buffers size to the #defined BLOCKSIZE set as our input drivers block size. *(To put this another way, since this is a dummy copy codec, we can cheat and know that this is always the right size.)*

To figure out what we need to do, let’s start at the top of the file (audio_thread.c), where we’ll find:

```c
#define BLOCKSIZE 44100
```

At the top of the function we created and initialized these variables:

```c
int blksize = BLOCKSIZE;      // raw input or output frame size
char *encBuf = NULL;         // pointer to encoded buffer
int encBufSize = 0;         // size of encoded buffer
```

Here’s the code you need to add to your file. Note, you may want to place this just after the malloc() for the outputBuffer which we’ve been using since our original audio lab.

```c
encBufSize = blksize;
encBuf = malloc(encBufSize);
```

Three notes about what you’ve just (or are about to) type in:

- At this stage, you can use malloc() to allocate this buffer since our local codecs will run on the ARM processor. Later we will use memory_contigAlloc() to allocate physically contiguous buffers which can be passed to remote codecs on the DSP. *(Actually, we could use the contiguous allocator function now so our code would be more portable, but that would mean less fun in a later lab exercise.)*

- While we could have hard-coded the encBufSize to BLOCKSIZE, this is much less flexible. Later when we switch to using a “real” codec, it’s better to specify the size with a variable (not a constant).
14. Replace the processing in the “while” loop with calls to audio encode and decode.

Within the while loop (in audio_thread.c), you should find the following code:

```c
read() from OSS audio device
memcpy()
write() to OSS audio device
```

You need to replace the memcpy() function call with two calls to encode/decode the audio:

```c
read()
encode_audio()
decode_audio()
write()
```

Again, since we are using a dummy copy codec, what the encoder returns is exactly what we put into it. Essentially, there’s just a memcpy() inside our codec. Because of this, we can just set the number of ‘encoded’ bytes it returns equal to the buffer size passed to it. (In a real codec, the encoder should return the actual size of the data after compression.)

```c
read(inputFd, inputBuffer, blksize)
numbytes = encBufSize;
encode_audio()
decode_audio()
write(outputFd, outputBuffer, blksize)
```

As a final hint, here is the prototype for our encode and decode “wrapper” functions. You might want to fill out these values before typing this in.

```c
int encode_audio(AUDENC_Handle encoderHandle, char *inputBuffer, int inputSize, char *outputBuffer, int *outputSizeByRef);
```

```c
int encode_audio(
    __encoderHandle_____________, // handle to encoder
    __inputBuffer_______________, // input buffer
    __blksize___________________, // size of input buffer
    __encBuf___________, // output buffer,
    __&numbytes_________________); // output buffer length (by ref)
```

Similarly, here’s the prototype of the decode wrapper function:

```c
int decode_audio(AUDDEC_Handle decoderHandle, char *inputBuffer, int inputSize, char *outputBuffer, int outputSize);
```

```c
int decode_audio(
    __decoderHandle___________, // handle to decoder
    __inputBuffer_______________, // buffer being input into decoder
    __blksize___________________, // size of buffer being put into decoder
    __decodedBuffer___________, // buffer output from decoder
    __&outputLength___________); // length of buff being output
```

Well, just one more hint. There are similar functions in the while loop for video_thread.c. If you get stuck, you might want to refer to it.
15. After `while` exits, add cleanup code for the buffers and instances added to this file.

There already is a “cleanup” section in `audio_thread_fx` with two calls to close the input, and output, audio drivers. You need to add code to clean the remaining resources we have added:

- Audio Encoder
- Audio Decoder
- Engine
- Encode Buffer (we discuss this in the next, in step 16)

Again, we have written “wrapper” functions for the instance close/delete functions. These are pretty simple function calls which only require a handle as an argument. Rather than giving you the code for all three, we demonstrate the simple – and advanced – ways to call these functions, letting you choose either method.

**Hint:** To view the prototypes for each of the three cleanup functions, please refer to their related header files:

```c
video_encoder.h
video_decoder.h
engine.h
```

**Hint:** Also, don’t forget that the engine cleanup must occur after all encoders and decoders associated with the engine have already been deleted.

**Hint:** If in doubt, look at how/where it was done in the associated video file.

Here’s the simple method of cleaning the encoder:

```c
audio_encoder_cleanup(encoderHandle);
```

And, here’s the fancy (and more debug friendly) way:

```c
/* Delete audio encoder */
if (initMask & AUDIOENCODERCREATED)
    if( audio_encoder_cleanup(encoderHandle) != AENC_SUCCESS ) {
        ERR("audio_encoder_cleanup failed\n");
        status = AUDIO_THREAD_FAILURE;
    }
```

Notice how this code only calls cleanup if the AUDIOENCODERCREATED bit in the `initMask` has been set, indicating that the setup function was completed correctly. If there is an error, this code (i.e. error printout) can help you isolate the problem.

16. Free the memory allocated to our intermediate buffer allocated in Lab step 0.

Since we allocated the buffer (i.e. encBuf) using malloc(), you can use the free() command to release the memory.

17. Save, then exit, the `audio_thread.c` file.
Modify audio_thread.h

18. **Extend the audio_thread_env structure, adding a way to pass the engine name from main() to audio_thread_fxn().**

This data structure is defined in the header file, and referenced by both main.c and audio_thread.c.

You may remember our chapter discussion strenuously suggesting that we pass the engine string name 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*.

```c
char *engineName;
```

Modify main.c

19. **Fix the initialization of the audio_env global variable.**

After changing the definition of audio_thread_env in step 18, our initialization is now incomplete. We recommend initializing the new element to NULL; in step 20, we will set it to the engine’s string name.

20. **Modify main.c so that it passes the engineName to our audio thread function.**

You just modified the audio_thread_env type definition in the step 18. In main.c, your code should fill-in the engine name element of the audio_env variable where this data structure is initialized.

What value should be used for the engine name? As with the string names for the audio encoder and decoder, you can find the engine string name in the .cfg file. In this lab, though, we have already defined a constant for you:

```c
#define  ENGINE_NAME = "encodedecode".
```

In main.c, just before you call pthread_create(), you need to set the engineName field in the audio_env structure to the name used by our Engine.

Build, install and Run Application

21. **Review the build script makefile_profile.mak, then build the program and test it out.**

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 and run gMake to build the application, then run it with:

```bash
make install
```
Lab09c_challenge1 — Control Thread

File Management

22. Create a new directory for this challenge.
   Make a new directory named lab09c_challenge1.

23. Copy the working lab09b_use_published_engine_av into lab09c_challenge1
    ```
    cp -R -f lab09b_use_published_engine_av/* lab09c_challenge1
    ```

Add Control Thread to main.c

24. Add a control thread to main.c.

   Modify lab09c_challenge1/app/main.c to add a control thread that scans an input string
   using scanf(), and, depending on the command, either creates a new audio or video thread or
   deletes the currently running audio or video thread using pthread_join().

   The machinery for this control thread can be copied directly from lab08d_challenge1. We
   just want to verify that we can instantiate and free audio and video codecs correctly when we
   create and delete the audio and video threads from the system.
DMAI (Digital Media App Interface) Library

Workshop Example – Lab06b with Audio Decode

```c
int audio_output_setup()
{
    inBuf = malloc()
    fopen("myaudio.dat")
    outBuf = malloc()
    engine_setup()
    audio_decoder_setup()
    fread data into inBuf
    while()
        decode_audio(inBuf, outBuf)
        write encBuf
        fread data into inBuf
}
```

- To simplify workshop labs, we originally created a set of "wrapper" functions to encapsulate driver and visa calls
- DMAI (digital media applications interface) library includes a more standardized set of functions to implement a similar set of wrapper functions

AAC Audio Decode Example

```c
Dmai_init();
/* Module and buffer to manage output to sound peripheral */
Sound_Handle  hSound  = Sound_create(&sAttrs);
Buffer_Handle hOutBuf = Buffer_create(Adec_getOutBufSize(hAd), &bAttrs);
/* Module and buffer to manage input from "myfile.aac" */
Loader_Handle hLoader = Loader_create("myfile.aac", &lAttrs);
Buffer_Handle hInBuf;
/* Module to manage audio decode */
Engine_Handle hEngine = Engine_open("myengine", NULL, NULL);
Adec_Handle hAd = Adec_create(hEngine, "aac", &params, &dynParams);
/* main algorithm */
Loader_prime(hLoader, &hInBuf);
while (1) {
    Adec_process(hAd, hInBuf, hOutBuf);
    Sound_write(hSound, hOutBuf);
    Loader_getFrame(hLoader, hInBuf);
    if (Buffer_getUserPtr(hInBuf) == NULL) break;
}
```

---

9 - 30 DaVinci/OMAP Workshop - Local Codecs - Using a Given Engine
Example of DMAI Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loader_create()</td>
<td>Creates a Loader</td>
</tr>
<tr>
<td>Loader_prime()</td>
<td>Prime the loader and obtain the first frame</td>
</tr>
<tr>
<td>Loader_getFrame()</td>
<td>Load the next frame</td>
</tr>
<tr>
<td>Adec_create()</td>
<td>Creates an Audio Decode algorithm instance</td>
</tr>
<tr>
<td>Adec_process()</td>
<td>Get the error code of the last failed operation.</td>
</tr>
<tr>
<td>Buffer_create()</td>
<td>Creates and allocates a contiguous Buffer</td>
</tr>
<tr>
<td>Buffer_delete()</td>
<td>Deletes and frees a contiguous Buffer</td>
</tr>
<tr>
<td>Buffer_getSize()</td>
<td>Get the size of a Buffer</td>
</tr>
</tbody>
</table>

- Library of functions to help ease use of Linux drivers and CE's VISA functions; esp. helpful when utilizing complex-useage algorithms like H.264
- Easier to port CE applications when using high-level functions like DMAI
- DMAI does not have anything to do with DMA (direct memory access)
- DMAI is now an open source project at: http://gforge.ti.com/gf/project/dmai
Codec Engine Functions Summary

Codec Engine Modules

Table 4-1 Codec Engine Modules

<table>
<thead>
<tr>
<th>Description</th>
<th>Module Abbreviation</th>
<th>Package Name</th>
<th>Header File(s)</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization function</td>
<td>CERuntime_</td>
<td>stdio.jsce</td>
<td>CERuntime.h</td>
<td>Section 4.2.1</td>
</tr>
<tr>
<td>Codec Engine Runtime</td>
<td>Engine_</td>
<td>stdio.jsce</td>
<td>Engine.h</td>
<td>Section 4.2</td>
</tr>
<tr>
<td>OS Abstraction Layer for memory</td>
<td>Memory_</td>
<td>stdio.jsce-osal</td>
<td>Memory.h</td>
<td>Section 4.6</td>
</tr>
</tbody>
</table>

Table 4-2 Codec Engine Modules

<table>
<thead>
<tr>
<th>Description</th>
<th>Module Abbreviation</th>
<th>Package Name</th>
<th>Header File(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Encoder Interface</td>
<td>VIDEOENC_</td>
<td>stdio.jsce</td>
<td>video.h</td>
</tr>
<tr>
<td>Video Decoder Interface</td>
<td>VIDEODEC_</td>
<td>stdio.jsce</td>
<td>video.h</td>
</tr>
<tr>
<td>Image Encoder Interface</td>
<td>IMAGEENC_</td>
<td>stdio.jsce</td>
<td>image.h</td>
</tr>
<tr>
<td>Image Decoder Interface</td>
<td>IMGDEC_</td>
<td>stdio.jsce</td>
<td>image.h</td>
</tr>
<tr>
<td>Speech Encoder Interface</td>
<td>SPEECHENC_</td>
<td>stdio.jsce</td>
<td>speech.h</td>
</tr>
<tr>
<td>Speech Decoder Interface</td>
<td>SPEECHDEC_</td>
<td>stdio.jsce</td>
<td>speech.h</td>
</tr>
<tr>
<td>Audio Encoder Interface</td>
<td>AUDIENCEC_</td>
<td>stdio.jsce</td>
<td>audio.h</td>
</tr>
<tr>
<td>Audio Decoder Interface</td>
<td>AUDIODEC_</td>
<td>stdio.jsce</td>
<td>audio.h</td>
</tr>
</tbody>
</table>

Core Engine Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERuntime_init</td>
<td>Initialize the Codec Engine</td>
</tr>
<tr>
<td>CERuntime_exit</td>
<td>Free CE memory. (Added starting with CE 1.20)</td>
</tr>
<tr>
<td>Engine_open()</td>
<td>Open an Engine.</td>
</tr>
<tr>
<td>Engine_close()</td>
<td>Close an Engine.</td>
</tr>
<tr>
<td>Engine_getCpuLoad()</td>
<td>Get Server’s CPU usage in percent.</td>
</tr>
<tr>
<td>Engine_getLastErrorCode()</td>
<td>Get the error code of the last failed operation.</td>
</tr>
<tr>
<td>Engine_getMem()</td>
<td>Get Engine memory usage.</td>
</tr>
<tr>
<td>Engine_getNumAlgs()</td>
<td>Get the number of algorithms in an Engine.</td>
</tr>
<tr>
<td>Engine_getAlgInfo()</td>
<td>Get information about an algorithm.</td>
</tr>
</tbody>
</table>
## Engine Return Values

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine_EOK</td>
<td>Success</td>
</tr>
<tr>
<td>Engine_EEXIST</td>
<td>Name does not exist</td>
</tr>
<tr>
<td>Engine_ENOMEM</td>
<td>Can't allocate memory</td>
</tr>
<tr>
<td>Engine_EDSPLOAD</td>
<td>Can't load the DSP</td>
</tr>
<tr>
<td>Engine_ENOCOMM</td>
<td>Can't create a communication connection to the DSP</td>
</tr>
<tr>
<td>Engine_ENOSERVER</td>
<td>Can't locate the Server on the DSP</td>
</tr>
<tr>
<td>Engine_ECOMALLOC</td>
<td>Can't allocate a communication buffer</td>
</tr>
</tbody>
</table>

### Engine_algInfo() - Return Values

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine_EOK</td>
<td>Success</td>
</tr>
<tr>
<td>Engine_EEXIST</td>
<td>There is no Engine with the given name</td>
</tr>
<tr>
<td>Engine_EINVAL</td>
<td>The algInfoSize field of the Engine_AlgInfo object passed to this function does not match the size of the Engine_AlgInfo object in the Codec Engine library</td>
</tr>
<tr>
<td>Engine.ENOTFOUND</td>
<td>The index of the algorithm is out of range</td>
</tr>
</tbody>
</table>

## Server Functions (Called by ARM-side Application)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine_getServer()</td>
<td>Get the handle to a Server</td>
</tr>
<tr>
<td>Server_getNumMemSegs()</td>
<td>Get the number of heaps in a Server</td>
</tr>
<tr>
<td>Server_getMemStat()</td>
<td>Get statistics about a Server heap</td>
</tr>
<tr>
<td>Server_redefineHeap()</td>
<td>Set base and size of a Server heap</td>
</tr>
<tr>
<td>Server_restoreHeap()</td>
<td>Reset Server heap to default base and size</td>
</tr>
</tbody>
</table>
### Memory Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory_contigAlloc()</td>
<td>Allocate physically contiguous blocks of memory</td>
</tr>
<tr>
<td>Memory_contigFree()</td>
<td>Free memory allocated by Memory_contigAlloc()</td>
</tr>
</tbody>
</table>
Local Codecs: Building an Engine

Introduction

In this chapter the steps to build a local engine will be considered.

Learning Outline

Topics covered in this chapter will include:

- Review of the Codec Engine Framework
- Review the Configuro tool
- Describe “Engine” configuration details
- (Optional) Build of an engine deliverable
Chapter Topics

Local Codecs: Building an Engine .................................................................................................................. 10-1

CE Framework Review ................................................................................................................................. 10-3

Engine Configuration Details ......................................................................................................................... 10-6

Lab 10 ............................................................................................................................................................ 10-9

Lab 10a_build_app_and_engine....................................................................................................................... 10-11

File Management ......................................................................................................................................... 10-11

Create RTSC configuration (.cfg) file........................................................................................................... 10-11

Update XDCPATH ......................................................................................................................................... 10-13

Build, Install, Run (and hopefully not need to debug...) ........................................................................... 10-14

Make it fail .................................................................................................................................................. 10-14

(Optional) Building an Engine Deliverable .................................................................................................. 10-16
CE Framework Review

The Application has access to both Local (ARM-side) xDM codecs and Remote (DSP-side) xDM codecs.

In Chapter 9 ...

- In the last chapter, we used an “engine” that someone else built
- To use a pre-built engine, we:
  - Link in the library that contains the engine
  - Call the engine/visa API
In Chapter 10 ...

Two goals for chapter 10:
1. Build our app and engine together
2. Optionally, build an engine (only) library to provide to others

Local Only Engine for ARM

Integrates:
1. Application
2. Codecs
3. Class functions (i.e VISA)
4. Engine functions
5. Codec table
6. O/S Abstraction Layer
Review : Using Configuro

- **Configuro** is a tool that helps users **consume/use packaged content**.

- **Packages needed:**
  - Codecs, Class functions (i.e VISA)
  - Engine functions
  - etc...

- **Configuro needs four other inputs to help it perform properly:**
  - XDCPATH – path to Configuro tool
  - Platform – e.g. ti.platforms.evmDM6446
  - Target – e.g. gnu.targets.MVArm9
  - .cfg – indicates which packages to include  ➔ Which packages should we include?
Engine Configuration Details

Local Only Engine for ARM

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.LINUX;
var vidDec = xdc.useModule('tto.codecs.viddec.VIDDEC');
var audDec = xdc.useModule('tto.codecs.auddec.AUDDEC');
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", []);
```
Local Only Engine for ARM

Example:

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.LINUX;

var vidDec = xdc.useModule('tto.codecs.viddec.VIDDEC');
var audDec = xdc.useModule('tto.codecs.auddec.AUDDEC');

var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
  {name: "video", mod: vidDec, local: true},
  {name: "audio", mod: audDec, local: true},
]);
```
Local Only Engine for ARM

Engine Configuration File (.cfg file)

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.LINUX;

var vidDec = xdc.useModule('tto.codecs.viddec.VIDDEC');
var audDec = xdc.useModule('tto.codecs.auddec.AUDDEC');

var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
    {name: "video", mod: vidDec, local: true},
    {name: "audio", mod: audDec, local: true},
];
```

Engine and Algorithm Names

```javascript
var myEng = Engine.create("myEngine", [  
    {name: "video", mod: vidDec, local: true},
    {name: "audio", mod: audDec, local: true},
];
```

Application Source File (app.c)

```c
CERuntime_init();
myCE = Engine_open("myEngine", myCEAttrs);
myAE = AUDDEC_create(myCE, "audio", params);

AUDDEC_control(myAE, ...);
AUDDEC_process(myAE, ...);
AUDDEC_delete(myAE);
Engine_close(myCE);
```
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

Build the engine and application in one step using gMake/Configuro

lab10a_build_app_and_engine
There are four different use-cases for interacting with the Codec Engine:

- **Linux application dev’l only** – Engine provided as .o lib
- **Engine provided by another in-house team or 3rd Party**
- **Modularity allows to focus on end-app**
- **Codes local or remote (don’t care)**

- **Similar to prev use-case, but you’re provided server containing remote codecs**
- **Simple (.cfg) method makes remote codecs easy to use**

- **Linux application dev’l – as well as Engine config (.cfg)**
- **Build Engine & app all-in-one**
- **While easy, user must know about plugins & .cfg files**
- **Local codecs only**

- **Similar to others from app engine producers persp.**
- **Building a server is like building a DSP executable (i.e. "out" file)**
- **Requires some embedded syst. dev’l experience**
Lab10a_build_app_and_engine

File Management

1. Clean your lab09b_use_published_engine_av project.

   cd ~/workshop/lab09b_use_published_engine_av/app
   make clean

2. Copy the lab09b_use_published_engine_av project to the Lab 10a directory.

   cd ~/workshop/lab09b_use_published_engine_av/app
   make clean

   cp –R -f ~/workshop/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 Lab 5d 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 library (-l) and include (-i) file
   path statements. (You might remember us discussing this feature of Configuro in Chapter 5.)

   cd ~/workshop/lab10a_build_app_and_engine/app
   cp -f ~/workshop/lab05d_standard_make/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 ~/workshop/lab10a_build_app_and_engine/app
   gedit app_cfg.cfg &

   Note: For most users, this command will create a new file named app_cfg.cfg, then
   open it within gedit. Although we didn’t specify this, if you happened to keep
   your similarly named .cfg file from Chapter 5, then you can just append the new
   content to your file.
5. **Import and configure the proper OSAL (O/S Abstraction Layer) required by local-only Linux engine.**

   Please refer back to the examples from the chapter to figure out how to import and configure the OSAL package.

   **Hint:** You will import all the packages using the `xdc.useModule()` method.

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:


   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"
   viddec copy: "video_decoder"
   audenc copy: "audio_encoder"
   auddec copy: "audio_decoder"
   ```

8. **When complete, save and close your config file.**
Update XDCPATH

9. Find the XDCPATH definition in your `makefile_profile.mak` file.

```
gedit makefile_profile.mak &
```

10. Configuro needs to know where to search for the packages you specified in your `.cfg` file.

You might remember we created a gMake variable (XDCPATH) which tells Configuro where to search. Currently, XDCPATH should be set to the following within your makefile:

```
export XDCPATH:=/home/user/rtsc_primer/examples;$(XDCROOT)
```

(Note, you can remove/replace the rtsc_primer examples as we won’t be using them at this point in time. Also, it’s recommended to list $(XDCROOT) last.)

In `makefile_profile.mak`, we need to specify the repositories (i.e. directories) which contain the packages we included with our `.cfg` file. Here’s a hint to help you find the required paths:

<table>
<thead>
<tr>
<th>Codec Engine</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codec Engine</td>
<td><code>${CE_INSTALL_DIR}/packages</code></td>
</tr>
<tr>
<td>‘copy’ codecs</td>
<td><code>${CE_INSTALL_DIR}/examples</code></td>
</tr>
<tr>
<td>req by codecs</td>
<td><code>${XDAIS_INSTALL_DIR}/packages</code></td>
</tr>
<tr>
<td>req by CE</td>
<td><code>${CMEM_INSTALL_DIR}/packages</code></td>
</tr>
<tr>
<td>req by CE</td>
<td><code>${FC_INSTALL_DIR}/packages</code></td>
</tr>
</tbody>
</table>
Build, Install, Run (and hopefully not need to debug…)

11. Make and install your program to the DVEVM target.

12. 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.

13. Execute the `app_debug.x470MV` 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.

14. Open your `makefile_profile.mak` and modify the XDCPATH statement – remove the CMEM directory reference – then save the file.

```
  gedit makefile_profile.mak &
```
15. 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/config/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/packages;/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/config/Main.xs", line 154
    
    "*/home/user/dvsdk_1_30_00_40/xdctools_3_10/packages/xdc/config/Main.xs", 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.

16. Repair your XDCPATH and re-test your solution.
Here are some rough notes about building a stand-alone engine that was provided for you (and that you used) in the Lab 9 exercises.

Note, we built our engine-only executable using the TI XDC tools. We will discuss how to use these tools in Chapter 12. The optional Lab09b contains the files needed to make the engine-only build using the these tools.

**Engine Deliverable (object file)**

- **Pkg.addExecutable** was intended for building executables
- **An ARM executable** (e.g. engine_app.x470MV) includes: application code + engine
- **You can create an Engine Deliverable (i.e. without an associated application) by using the linker's –r flag**

In other words, for an Engine Deliverable, we build in the same way, but using a partial link step, and without linking an application

While stand-alone engine libraries can be built using gMake files, we used the TI XDC tools to create our stand-alone engine library. If you are interested, try looking over the RTSC build description files to see how we accomplished this.

- _dummy.c files
- engine.cfg
- package.xdc
- config.bld
- package.bld
Build Script for Engine Archive Deliverable

```javascript
/* ======== package.bld ========*/

var appName = "engine_app";
var targ = MVArm9_o;
var sources = [ "audio_decoder_dummy.c",
               "audio_encoder_dummy.c", ... ];

Pkg.addExecutable( appName, targ, "ti.platforms.evmDM6446",
{  
  cfgScript: "engine.cfg",
  profile: "debug",
  lopts: "-Wl,-r -nostdlib",
}).addObjects( sources );

◆ Specify MVArm9_o target to indicate we are building an Arm engine object file
◆ Application source files replaced with dummy routines
◆ –Wl,r indicates options are to be used by the Linker
  • –r option provides partial link into object file (i.e. builds a relocatable object file)
◆ –nostdlib : do not use standard C initialization routines; rather, they will be included by the final end-application build

Dummy Functions

audio_decoder_dummy.c

```c
void __audio_decoder_dummy(void) {
    // declarations

decoderHandle = AUDDEC_create(engineHandle, "invalid", NULL);
AUDDEC_process(decoderHandle, &inBufDesc, &outBufDesc, &inArgs, &outArgs);
AUDDEC_control(decoderHandle, XDM_GETSTATUS, &dynParams, &status);
AUDDEC_delete(decoderHandle);
}
```

◆ Linker will not link functions from libraries unless they are called from within a source file.
◆ Dummy functions exercise the 4 basic VISA API to ensure all functions are linked into engine deliverable
◆ Some engine providers will provide diagnostic routines instead of dummy functions for added functionality.
*** Why on earth are you staring at a blank page? ***
Remote Codecs: Given a DSP Server

Introduction

In this chapter the steps required to use a given DSP server will be examined

Learning Outline

In this chapter the following topics will be presented:

- Remote codec calls
- Server management layer
- Internal mechanisms within the Codec Engine
- Modifying the Engine to use a Remote Server
Chapter Topics

Remote Codecs: Given a DSP Server ........................................................................................................11-1

Remote Codec Calls ..................................................................................................................................11-3

Server Management Layer (VISA functions) ............................................................................................11-7

Priorities on the DSP Server ....................................................................................................................11-8

Modifying the Engine to use a Remote Server ..........................................................................................11-10

Contiguous Memory Functions ..................................................................................................................11-11

Lab 11 .........................................................................................................................................................11-12

Lab 11a – Using a Published Server ..........................................................................................................11-13

Lab 11b_challenge1 – Virtual and Physical Addresses ..............................................................................11-16
Remote Codec Calls

The Application has access to both Local (Arm-side) xDM codecs and Remote (DSP-side) xDM codecs.
Remote Codec Calls

Local Procedure Call

Master Thread / App code

```
... VIDENC_process( a, b, c );
...
```

```
process( a, b, c ) {
...
...
}
```

ARM

Master Thread / App code

Remote Procedure Call “RPC”

ARM Stack

App locals

```
VIDENC_process( a, b, c );
```

```
process( a, b, c ) {
...
...
}
```

ARM

DSP Stack

locals

DSP

- In addition to packaging variables for transport to the DSP, the stub is responsible for any necessary address translation of ptrs
Remote Procedure Call “RPC”

In addition to packaging variables for transport to the DSP, the stub is responsible for any necessary address translation of pointers.
It is actually the VISA Functions which are executed as remote procedure calls on the server, as shown above.

But we already learned that the VISA functions utilize Engine SPI, so…

The server contains Engine SPI just as the client does
These SPI are identical, except for the OSAL layer, which allows the client Engine to communicate with Linux and the server Engine to communicate with BIOS.
Server Management Layer (VISA functions)

**VISA create/delete**

- **RMS** only runs to create or delete codec instances

**VISA process/control**

- For runtime calls (_process or _control), the CE goes directly to the specific instance handling the codec processing request
- Cutting out the RMS minimizes runtime latency
Priorities on the DSP Server

When multiple codecs need to execute, who goes first?

- Codec Engine executes DSP algorithm tasks on a priority basis;
  Using priority-based scheduling on the DSP Server, the Audio thread completes before the Video thread, as intended
- If the DSP server worked on a FIFO basis (first-in first-out), then the ARM threads might end up with a priority inversion problem

Priority Inversion from FIFO Server Response

The higher priority Audio thread (on the ARM) completes later than the lower priority Video thread
CE implements DSP priorities by running each codec in its own prioritized thread (BIOS task)
You specify to CE the DSP priorities via the server's config (.cfg) file
Modifying the Engine to use a Remote Server

### Changing Engine to use a Server

**Engine.cfg Configuration File**

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.DSPLINK_LINUX;

var audEnc1 = xdc.useModule('codecs.audenc1.AUDENC1');
var audEnc2 = xdc.useModule('codecs.audenc2.AUDENC2');

var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
  {name: "myEnc1", mod: audEnc1, local: false},
  {name: "myEnc2", mod: audEnc2, local: false},
]);

myEng.server = "./myServer.x64P";
```

- Set Codec properties to local=false to tell them to remotely on the DSP server.
- Warning: The module and codec names must match between engine and server!

- If using remote codecs (local: false), specify the DSP executable file name in the engine config.
- When Engine_open() is called, the ARM-side engine automatically loads the server image onto the DSP.

Better yet …

### Create Engine from Server

**Old Method**

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.DSPLINK_LINUX;
var audEnc1 = xdc.useModule('codecs.audenc1.AUDENC1');
var audEnc2 = xdc.useModule('codecs.audenc2.AUDENC2');
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
  {name: "myEnc1", mod: audEnc1, local: false},
  {name: "myEnc2", mod: audEnc2, local: false},
]);
myEng.server = "./myServer.x64P";
```

**New Method**

```javascript
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.DSPLINK_LINUX;
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.createFromServer(  
  "myEngine", // Engine name (as referred to in the C app)  
  ".MyServer.x64P", // Where to find the .x64P exe, inside the server's package  
  "tto.servers.example" ); // Server's package name
myEng.server = ".MyServer.x64P"; // Loc'n of server exe at runtime, relative to .x470MV program;  
  // only needed if not found in the same folder as .x470MV
```

**New Method is:**
- Shorter (i.e. easier)
- Less error prone
- Configuro can extract additional info from package metadata (e.g. memory map)
Contiguous Memory Functions

Allocating Contiguous Memory Buffers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory_contigAlloc()</td>
<td>Allocate physically contiguous blocks of memory</td>
</tr>
<tr>
<td></td>
<td>void Memory_contigAlloc (unsigned int size, unsigned int align)</td>
</tr>
<tr>
<td>Memory_contigFree()</td>
<td>Free memory allocated by Memory_contigAlloc</td>
</tr>
<tr>
<td></td>
<td>bool Memory_contigFree (void *addr, unsigned int size)</td>
</tr>
</tbody>
</table>

Linux memory allocations may actually be non-contiguous in physical memory – that's the advantage of relying on an MMU (Memory Management Unit). Since most DSPs do not use an MMU, any buffers passed to the DSP must be physically contiguous. The Linux CMEM driver (found in the linuxutils package) performs contiguous buffer allocations.

Example / Comparison:

<table>
<thead>
<tr>
<th>Standard C syntax</th>
<th>Using Memory_contig() functions</th>
</tr>
</thead>
</table>
| ```
#include <stdlib.h>
#define SIZE 128

a = malloc(SIZE);
a = {...};
use buffer...
free(a);```                                                                  | ```
#include <ti/sdo/ce/osal/Memory.h>
#define SIZE 128

a = Memory_contigAlloc(SIZE, ALIGN);
a = {...};
use buffer...
Memory_contigFree(a, SIZE);```                                                  |

Summary of Labs 9-12

 Codec Engine - Use Cases

- There are four different use-cases for interacting with the Codec Engine
- Here's a quick description, along with a reference where it's covered

<table>
<thead>
<tr>
<th>Lab 9</th>
<th>Lab 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Linux application dev'l only - Engine provided as .o lib
- Engine provided by another in-house team or 3rd Party
- Modularity allows to focus on end-app
- Codecs local or remote (don't care)

- Linux application dev'l – as well as Engine config (.cfg)
- Build Engine & app all-in-one
- While easy, user must know about pkgs & .cfg files
- Local codecs only

<table>
<thead>
<tr>
<th>Lab 11</th>
<th>Lab 12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Similar to prev use-case, but you're provided server containing remote codecs
- Simple (.cfg) method makes remote codecs easy to use

- Similar to others from app/engine producers persp.
- Building a server is like building a DSP executable (i.e. *.out* file)
- Requires some embedded syst. dev'l experience
Lab 11

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 the next lab, you will build the 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 ~/workshop/lab11a_publish_server
   cp -R -f ~/workshop/lab10a_build_app_and_engine/* .

2. List the contents of the “server” folder, notice the DSP executable files.

   In addition to the app folder that you copied over from lab10a_build_app_and_engine, there is an additional folder – named server – which is provided for you. The server folder contains a “published” server executable image.

3. Change into the app directory of lab11a_publish_server.

Modify Engine Configuration File (app_cfg.cfg)

4. Open the config file (app_cfg.cfg) and update the OSAL runtime environment to include DSPLINK.

   osal.runtimeEnv = _________________

5. 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.

```
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),
});
```

Replace

Three hints:

- Find the name of the server by looking in the “server” folder.
- The name of the server package is “server”. (We will see how this package name is defined in the next chapter.)
- Because the server .x64P and the ARM .x470MV applications will both be executed from the same /opt/workshop directory, the engine’s server property does not need to be specified.
Changes needed to makefile_profile.mak

Two changes are required to get our build script up-to-date.

6. **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. Since the server package is called “server”, we need to look for it’s parent directory (the one that contains the folder named “server”). (Note, you should use lab11a rather than 12a.)

   ```
   # ---
   100  XDCROOT := $(XDC_INSTALL_DIR)
   101  CONFIGURO := $(XDCROOT)/xs xdc.tools.configuro
   102  LAB12a_DIR := /home/user/workshop/lab12a_build_server
   103  LAB_DIR := $(LAB12a_DIR)
   104  export XDCPATH := $(LAB_DIR);$(CONFIG_BLD_PATH);$(CE_INSTALL)
   105  # ---
   ``

   - We also need to specify the paths to our codecs, but since the ‘copy’ codecs are located in the Codec Engine examples directory, we already have that path in our XDCPATH.
   - It is recommended that you place the $(XDCROOT) variable at the end of the $(XDCPATH) string command.

7. **Modify the “install” rule so that it also copies the server executables.**

   **Under the install command:** @cp $^ $(EXEC_DIR)

   **Add the command:** @cp ../server/server.x64P $(EXEC_DIR)

   Note, you may also want to modify your “clean” rule to also remove it from the $(EXEC_DIR).
Modify audio and video "_thread.c" files

8. Convert memory allocations so they create contiguous buffers.

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, any buffers passed by the ARM to the DSP need to be allocated as physically contiguous buffers. This is best done using the Codec Engine’s memory functions:

```c
ptr Memory_contigAlloc(unsigned int size, unsigned int align);
bool Memory_contigFree(void *addr, unsigned int size);
```

Note that for alignment, you must specify a power of two, although it is most common to use an alignment of 8 bytes or greater, as this may improve the DSP’s efficiency by allowing double-word aligned loads.

In both the audio_thread.c and video_thread.c files we must convert malloc() functions into Memory_contigAlloc() functions; similarly, replace the matching “free” functions.

**Hint:** You can search on “malloc” and “free” in your code to find all occurrences. There should be one malloc and one free call in video_thread.c and three malloc calls and three free calls in audio_thread.c.

For more information on the memory functions (as well as VISA functions), you can go to the Codec Engine documentation by using:

```bash
cd /home/user/dvsdk_2_00_00_22/codec_engine_2_23_01
mozilla file://$(pwd)/codec_engine_2_23_01_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.

9. Reference the header file for the contiguous memory allocation functions.

In both audio_thread.c and video_thread.c files, add the reference to the memory library:

```c
#include <ti/sdo/ce/osal/Memory.h>
```
Build and run

10. Build and install the application.

11. Don’t forget to load the DSPLINK and CMEM modules.

   ```bash
   loadmodules.sh
   ```

   The script loads two drivers:
   - CMEM driver performs the contiguous memory allocations we just added.
   - DSPLINK driver provides inter-processor communications between the ARM and DSP.

   If the load and unload scripts are not in your target DVEVM’s `/opt/workshop` directory, you can run the script in `~/workshop/lab00a_install_scripts` to copy them over for you.

12. Execute the app_DEBUG.x470MV application.

Lab 11b_challenge1 – Virtual and Physical Addresses

The pointer value that `Memory_contigAlloc()` returns is a virtual memory address and does not correlate to a ‘real’ physical address in memory. Often, the ability to display the physical address is important when debugging, especially since the DSP uses only physical addresses.

While the Codec Engine automatically translates virtual-to-physical addresses when passing between the processors, providing the physical address can still be useful for debugging purposes.

13. Begin by copying the files to our new working directory.

   ```bash
   cp -R -f lab11a_publish_server/* lab11b_challenge1_physical
   ```

14. Add debug feedback to display the physical addresses of memory allocated `Memory_contigAlloc()`.
Remote Codecs: Building a DSP Server

Introduction

In the previous chapter you built an engine that contained a DSP Server. This DSP Server was provided for you, as if you had purchased one from a TI Third Party or obtained it from another team within your company.

In this chapter, we examine the steps required to build a DSP Server. Once built, you will incorporate this into an engine as was done in the previous chapter.

As a final note, why are these codecs referred to as “Remote Codecs”? You might remember, from the application’s perspective, Codecs running on the DSP (i.e. contained in the DSP Server) are considered to be running remote from the Linux/ARM application.

Learning Objectives

- Review basic codec engine architecture
- Describe the files required to build a DSP Server
- Use the Codec Engine’s DSP Server Wizard
- Describe how CE/DSKT2 manages xDAIS algos
Chapter Topics

Remote Codecs: Building a DSP Server ...........................................................................................................12-1

Codec Engine Review ........................................................................................................................................12-3

What Makes Up a DSP Server ........................................................................................................................12-4

DSP Server Wizard .............................................................................................................................................12-8

Building a Server – Config (cfg) Files ...........................................................................................................12-10

server.cfg ..................................................................................................................................................12-10

codec.cfg ..................................................................................................................................................12-11

Memory – Scratch Groups ...............................................................................................................................12-13

Back to server.cfg … and Memory Setup ......................................................................................................12-14

Sidebar – Minimum Scratch Group Allocations ..........................................................................................12-18

(Optional) A Short Course in XDC ................................................................................................................12-19

Lab 12 ............................................................................................................................................................12-22

Lab12a_build_server ....................................................................................................................................12-23

Lab12b_real_h264 ..........................................................................................................................................12-27

(Optional) Lab12c_h264_record .....................................................................................................................12-38

(Optional) Lab12d_h264_playback ..................................................................................................................12-40

Discussion: A-V Sync .......................................................................................................................................12-41
Codec Engine Review

CE Framework Details

- Looking at both sides of the ARM + DSP, we notice the similarities
- The app at the top of the ARM (client) side, and its proxies on the DSP side; the CE and VISA layers in the middle, as well as the RPC blocks (stub/skeleton) and finally the codecs (and OSAL) at the bottom.

The DSP Server

- The DSP Server is a stand-alone DSP executable file
- Traditionally, a DSP executable is called a “.out” file, with DaVinci, we chose to use the extension “.x64P”
What Makes Up a DSP Server

Building a Server

- What source packages might be needed to build a server?

A Server’s Source Packages

- Codecs
- Server Mgmt
  - ti.sdo.ce.server
- Engine SPI
  - ti.sdo.ce.engine
- OS Abstraction
  - ti.sdo.ce.osal
- Plus a couple others …
What Makes Up a DSP Server

Building a Server

Source Files
- main.c
- BIOS Config (.tcf)

Build Instructions
- XDC Build Tool

Executable File
- Package.x64P

What source packages might be needed to build a server?
- Why is main.c needed?

main.c

```c
#include <xdc/std.h>
#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/trace/gt.h>

static GT_Mask gtMask = {0,0}; // trace info: mod name, mask

/* ======= main ======== */
Void main (Int argc, Char * argv []) {
    /* Init Codec Engine */
    CERuntime_init();

    /* Init trace */
    GT_init();

    /* Create a mask to allow a trace-print welcome message below */
    GT_create(&gtMask, "codec_unitserver");

    /* ...and initialize all masks in this module to "on" */
    GT_set("codec_unitserver=01234567");

    GT_0trace(gtMask, GT_4CLASS, "main> Welcome to DSP server's main()>ln");
}
```

- Most importantly, you need to call CERuntime_init()
- This file was created by the Codec Engine's DSP Server Wizard
What Makes Up a DSP Server

Building a Server

- Source Files
- Packages
- main.c
- BIOS Config (.tcf)

- Build Instructions
- XDC Build Tool
- $> XDC

✓ What source packages might be needed to build a server?
✓ Why is main.c needed?
◆ Why include a BIOS configuration file (.tcf)?

BIOS Config File (.tcf)

```c
var mem_ext = [
{ comment: "Ext mem for DSP algo heap",
  name: "DDRALGHEAP",
  base: 0x88000000, // 128MB
  len: 0x07A00000, // 122MB
  space: "code/data"
},
{ comment: "Ext mem region for DSP code/data",
  name: "DDR",
  base: 0x8FA00000, // 250MB
  len: 0x00400000, // 4MB
  space: "code/data"
},
...]
```

◆ What .tcf does:
  - Defines DSP memory map
  - Init DSP reset/interrupt vectors
  - Creates/initiates various other DSP/BIOS data/prog objects

◆ No predefined .tcf filename, but name must match the .cfg file we discuss later in the chapter, for example:
  - server.tcf
  - server.cfg

DaVinci/OMAP Workshop - Remote Codecs: Building a DSP Server
Building a Server

Source Files

- main.c
- BIOS Config (.tcf)

Build Instructions

Executable File

$> XDC

What source packages might be needed to build a server?

Why is main.c needed?

Why include a BIOS configuration file?

What RTSC files are required?

RTSC files created by DSP Server Wizard

- package.xdc - name, version, dependencies
- package.xs  - build time interfaces
- package.bld - build instr's (gcc, options, profiles, etc.)
- server.cfg  - specifies inputpkg's
DSP Server Wizard

Build DSP Server Using Codec Engine Wizard

- Choose Platform:
  - ti.platforms.evnm6445
  - ti.platforms.evnm6450
  - ti.platforms.evnm6467
  - ti.platforms.evnm64P137

- Package Name
- Output Location (repository)
- Algorithms/Codecs in your DSP server

- By default, wizard finds all algo’s on your XDCPATH
- Set Pkg Search Path allows you to vary which folders to search for algorithms

Build DSP Server Using Codec Engine Wizard

- Pick “algo name” used in VIDDEC_create()
- Choose priority via slider
- By default, groupings follow priority
- Override defaults for Stack Location and Stack Size:

- video_decoder

DaVinci/OMAP Workshop - Remote Codecs: Building a DSP Server
Files for DSP Server

- **Generic main.c**
  You probably won't need to change it

- **Empty link.cmd**
  In case you need special linker options

- **RTSC Package files**
  - How to build the package
  - Defines package’s contents
  - Methods XDC tools can call

- **BIOS config file server.tcf**
  Defines memory and other objects as it would for any BIOS program

- **RTSC config file server.cfg**
  - Platform default server configuration
  - Well commented
  - Name matches .tcf filename

- **RTSC config file codec.cfg**
  - Included by server.cfg
  - Algorithm specific package info
Building a Server – Config (.cfg) Files

RTSC Config File (.cfg)

The RTSC configuration file (.cfg) allows you to specify which packages to include, as well as setting their configuration options.

1. General Setup
   - Add OSAL module
   - Add server module
   - Include codec.cfg

2. Memory Setup (DSKT2 module)
   - Associate IALG memory types with available target memory
   - Specify default scratch group memory sizes

3. DMA Setup (DMAN3) *(discussed in chapter 14)*
   - Specify number of DMA channels DMAN3 may manage
   - Specify number of TCCs and Params DMAN3 can offer to algos

4. CODEC Setup:
   - Add module reference for each codec
   - Configure properties for each codec thread:
     - Thread priority
     - Stack size
     - Stack memory location

server.cfg

Setup OSAL (server.cfg)

```c
/* ================ set up OSAL ===============*/
var osalGlobal = xdc.useModule('ti.sdo.ce.osal.Global');
osalGlobal.runtimeEnv = osalGlobal.DSPLINK_BIOS;

/* ============== Server Configuration ============== */
var Server = xdc.useModule('ti.sdo.ce.Server');

/* The server’s stackSize. More than we need... but safe. */
/* And, the servers execution priority */
Server.threadAttrs.stackSize = 16384;
Server.threadAttrs.priority = Server.MINPRI;

/* Import the algorithm's configuration */
utils.importFile("codec.cfg");
```
**Server Configuration (server.cfg)**

```javascript
/* ================ set up OSAL ================ */
var osalGlobal = xdc.useModule('ti.sdo.ce.osal.Global');
osalGlobal.runtimeEnv = osalGlobal.DSPLINK_BIOS;

/* ============== Server Configuration ============== */
var Server = xdc.useModule('ti.sdo.ce.Server');

/* The server's stackSize.  More than we need... but safe. */
/* And, the servers execution priority */
Server.threadAttrs.stackSize = 16384;
Server.threadAttrs.priority = Server.MINPRI;

/* Import the algorithm's configuration */
utils.importFile("codec.cfg");
```

**Use Codec Modules (codec.cfg)**

```javascript
/* =========== Use various codec modules =========== */
var VIDDEC_COPY =
    xdc.useModule('codecs.viddec_copy.VIDDEC_COPY');
var VIDENC_COPY =
    xdc.useModule('codecs.videnc_copy.VIDENC_COPY');

/* ============== Server Configuration ============== */
Server.algs = [
    {
        name: "viddec_copy", mod: VIDDEC_COPY, threadAttrs: {
            stackSize: 1024, stackMemId: 0, priority: 1
        },
    },
    {
        name: "videnc_copy", mod: VIDENC_COPY, threadAttrs: {
            stackSize: 1024, stackMemId: 0, priority: 1
        },
    },
];
```
Configure Server Algo’s (codec.cfg)

```javascript
/* =========== Use various codec modules =========== */
var VIDDEC_COPY =
    xdc.useModule('codecs.viddec_copy.VIDDEC_COPY');
var VIDENC_COPY =
    xdc.useModule('codecs.videnc_copy.VIDENC_COPY');
/* ============== Server Configuration ============== */
Server.algs = [
    {name: "viddec_copy",
    mod: VIDDEC_COPY,
    threadAttrs: {
        stackSize: 1024,
        stackMemId: 0,
        priority: 1
    }},
    {name: "videnc_copy",
    mod: VIDENC_COPY,
    threadAttrs: {
        stackSize: 1024,
        stackMemId: 0,
        priority: 1
    }}
];
```

Codec Config Parameters

```javascript
var V_COPY = xdc.useModule('codecs.viddec_copy.VIDDEC_COPY');
var Server = xdc.useModule('ti.sdo.ce.Server');

Server.algs = [
    {
        name: "viddec_copy",
        mod: V_COPY,
        threadAttrs: {
            stackSize: 1024,
            stackMemId: 0,
            priority: Server.MINPRI + 1
        },
        groupid: 1
    }
];
```

- By default, the Codec Engine automatically matches algorithms scratch memory ID to their thread priority, to help guaranteeing safe operation. (See wiki.davincidsp.com topic for exact details.)
Memory – Scratch Groups

Groups of Scratch Memory

Low Priority Algos (Priority = 0)
- Algo A1
- Algo A2

High Priority Algos (Priority = 4)
- Algo B1
- Algo B2
- Algo B3

- Scratch memory makes sense – without it, our mem req wouldn’t have fit on slide
- General rule – don’t allow differing priorities to share same scratch resources
- Priority ≠ groupid, though it’s a common to assign them that way
Back to server.cfg … and Memory Setup

RTSC Config File (.cfg)
The RTSC configuration file (.cfg) allows you to specify which packages to include, as well as setting their configuration options

1. General Setup
   - Add OSAL module
   - Add server module
   - Include codec.cfg

2. Memory Setup (DSKT2 module)
   - Associate IALG memory types with available target memory
   - Specify default scratch group memory sizes

3. DMA Setup (DMAN3) *(discussed in chapter 14)*
   - Specify number of DMA channels DMAN3 may manage
   - Specify number of TCCs and Params DMAN3 can offer to algos

4. CODEC Setup:
   - Add module reference for each codec
   - Configure properties for each codec thread:
     - Thread priority
     - Stack size
     - Stack memory location

TCF File : Target Memory Definitions

```javascript
var mem_ext = [
    { comment: "algo heap sent to external memory",
      name: "DDRALGHEAP",
      base: 0x88000000, // 128MB
      len: 0x07A00000, // 122MB
      space: "code/data"
    },
    { comment: "application code and data sent to external memory",
      name: "DDR",
      base: 0x8FA00000, // 250MB
      len: 0x00400000, // 4MB
      space: "code/data"
    },
    { comment: "DSPLINK code and data routed to external memory",
      name: "DSPLINKMEM",
      base: 0x8FE00000, // 254MB
      len: 0x00100000, // 1MB
      space: "code/data"
    },
    { comment: "reset vector routed to external memory",
      name: "RESET_VECTOR",
      base: 0x8FF00000,
      len: 0x00000080,
      space: "code/data"
    }];
```
### DM6446 EVM Memory Map

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1DSRAM</td>
<td>Level 1 On-Chip SRAM (0 waitstate)</td>
</tr>
<tr>
<td></td>
<td>• 16K cache</td>
</tr>
<tr>
<td></td>
<td>• 64K sram</td>
</tr>
<tr>
<td></td>
<td>(All 64K of L2 is config’d as cache)</td>
</tr>
<tr>
<td>Linux</td>
<td>Set via Linux command line to MEM=120M</td>
</tr>
<tr>
<td>CMEM</td>
<td>Set with CMEM insmod command</td>
</tr>
<tr>
<td>DSP Heap (DDRALGHEAP)</td>
<td>Set in BIOS Textual Config (.tcf) file</td>
</tr>
<tr>
<td>App Prog (DDR)</td>
<td>Set in BIOS Textual Config (.tcf) file</td>
</tr>
<tr>
<td>DSP Link</td>
<td>Set in BIOS Textual Config (.tcf) file</td>
</tr>
<tr>
<td>Reset/Int</td>
<td>Set in BIOS Textual Config (.tcf) file</td>
</tr>
</tbody>
</table>

### xDAIS Memory Spaces

- xDAIS specifies a variety of Memory Spaces an algorithm can request
- These various space names matched the h/w capabilities on older proc's

#### xDAIS Memory Spaces

- "Dual Access" RAMs:
  - DARAM0
  - DARAM1
  - DARAM2
- "Single Access" RAMs:
  - SARAM0
  - SARAM1
  - SARAM2
- External Data: ESDATA
- Internal/External Pgm:
  - IPROG
  - EPROG
- Heap for DSKT2: DSKT2_HEAP

#### With hindsight, it might have been better to name the spaces: ‘Space 1’, ‘Space 2’, etc.

#### Use these different spaces as a generic means of specifying diff parts of memory

#### For example, algo vendors may request a memory space, then indicate in the algo docs how to match the space to real memory:

- Algo1 asks for block of memory from DARAM0 and another from SARAM0
- In the documentation, they could state:
  - Map DARAM0 to L1D SRAM; and,
  - Map SARAM0 to L2 (or L1D) SRAM

The algo must run no matter which memory it is granted, though, to obtain spec'd performance, you must grant the requested memory spaces.
Connecting Spaces to Memory (i.e. what to give DSKT2)

- **L1DSRAM**
  - Level 1 On-Chip SRAM (0 waitstate)
  - 16K cache
  - 64K sram
  - (All 64K of L2 is config’d as cache)
  - Set via Linux command line to MEM=120M

- **Linux**
  - 120M

- **CMEM**
  - 128M
  - Set with CMEM insmod command

- **DDRALGHEAP**
  - 250M
  - Set in BIOS Textual Config (.tcf) file

- **App Prog**
  - 254M
  - Set in BIOS Textual Config (.tcf) file

- **DSP Link**
  - 255M
  - Set in BIOS Textual Config (.tcf) file

- **Reset**
  - 256M

---

- **“Dual Access” RAMs:**
  - DARAM0
  - DARAM1
  - DARAM2

- **“Single Access” RAMs:**
  - SARAM0
  - SARAM1
  - SARAM2

- External Data:
  - ES_DATA

- Internal/External Pgm:
  - IPROG
  - EPROG

- Heap for DSKT2:
  - DSKT2_HEAP

---

- **Level 1 On-Chip SRAM (0 waitstate):**
  - 16K cache
  - 64K sram
  - (All 64K of L2 is config’d as cache)

- **Set via Linux command line to MEM=120M**

- **Set with CMEM insmod command**

- **Set in BIOS Textual Config (.tcf) file**

- **Set in BIOS Textual Config (.tcf) file**

- **Set in BIOS Textual Config (.tcf) file**
Acts as a warehouse for Memory resources

- You configure the DSKT2 module with available memory resources (in server.cfg file)
- Algorithms “check out” memory from the DSKT2 module at runtime when they are instantiated

```javascript
var DSKT2 = xdc.useModule('ti.sdo.fc.dskt2.DSKT2');
DSKT2.DARAM0   = "L1DSRAM";
DSKT2.DARAM1   = "L1DSRAM";
DSKT2.DARAM2   = "L1DSRAM";
DSKT2.SARAM0   = "L1DSRAM";
DSKT2.SARAM1   = "L1DSRAM";
DSKT2.SARAM2   = "L1DSRAM";
DSKT2.ESDATA   = "DDRALGHEAP";
DSKT2.IPROG    = "L1DSRAM";
DSKT2.EPROG    = "DDRALGHEAP";
DSKT2.DSKT2_HEAP = "DDR";
```

- Algorithms req mem from the DSKT2 (i.e. CE) using xDAIS/iALG identifiers (DARAM0, DARAM1, etc)
- Those identifiers are tied to system named memory objects
- The memory names must match those described in the BIOS textual configuration file (server.tcf)
Sidebar – Minimum Scratch Group Allocations

Is Algorithm Creation Order Important?

Scenario 1

<table>
<thead>
<tr>
<th>Algo</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>4K</td>
</tr>
<tr>
<td>B2</td>
<td>2K</td>
</tr>
<tr>
<td>B3</td>
<td>3K</td>
</tr>
<tr>
<td>Total</td>
<td>4K</td>
</tr>
</tbody>
</table>

2K and 3K scratch req’s fit inside the first 4K alloc. Total of 4K scratch mem.

Scenario 2

<table>
<thead>
<tr>
<th>Algo</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>3K</td>
</tr>
<tr>
<td>B2</td>
<td>2K</td>
</tr>
<tr>
<td>B3</td>
<td>4K</td>
</tr>
<tr>
<td>Total</td>
<td>7K</td>
</tr>
</tbody>
</table>

In this case, largest alloc came last. Thus, when requested, 4K block won’t fit. A new 4K alloc must be done.

- Yes, order of algo creation can be important when sharing scratch memory
- Codec with largest scratch requirements (from a given pool) should be alloc’d 1st
- Though, this is better solved using another config parameter ... see next slide ...

DSKT2 – Minimum Scratch Group Memory Alloc’s

```javascript
var DSKT2 = xdc.useModule('ti.sdo.fc.dskt2.DSKT2');

DSKT2.DARAM0 = "L1DSRAM";
DSKT2.DARAM1 = "L1DSRAM";
DSKT2.DARAM2 = "L1DSRAM";
DSKT2.SARAM0 = "L1DSRAM";
DSKT2.SARAM1 = "L1DSRAM";
DSKT2.SARAM2 = "L1DSRAM";
DSKT2.ESDATA = "DDRALLOCHEAP";
DSKT2.IPROG = "L1DSRAM";
DSKT2.EPROG = "DDRALLOCHEAP";
DSKT2.DSKT2_HEAP = "DDR";

DSKT2.DARAM_SCRATCH_SIZES = [ 65536, 1024, 0, 0, /* ... */ 0 ];
DSKT2.SARAM_SCRATCH_SIZES = [ 65536, 1024, 0, 0, /* ... */ 0 ];

if( this.prog.build.profile == "debug" )
    DSKT2.debug = true;
```

- Algorithms req mem from the DSKT2 (i.e. CE) using xDAIS/iALG identifiers (DARAM0, DARAM1, etc)
- Those identifiers are tied to system named memory objects
- The memory names must match those described in the BIOS textual configuration file (server.tcf)
- The size for each scratch memory pool is set in an array.
- The first element is for scratch memory pool id = 0, the second for pool id=1, etc.
Moreover, unlike other build tools,

- **XDC can generate code** when provided with a configuration script. This is useful when combining package(s) into a new executable.
- This feature significantly simplifies building complex executables (like DaVinci Engines and DSP Servers) which will be explored later.
- **XDC is a superset of Configuro** – while Configuro can consume pkg’s, XDC can both consume and create packages.

Four files comprise XDC build instructions:

- **package.xdc**: names the RTSC package
- **package.xs**: defines build-time interfaces for modules
- **package.bld**: package build instructions
- **config.bld**: defines build targets and platforms (which are ref’d by package.bld)
package.xdc (2 examples)

Simple

```
// package.xdc for tto workshop app
package app { }
```

Using more RTSC Features

```
requires ti.sdo.ce.video;
// package.xdc for the video_copy example
package servers.video_copy [1, 0, 0] {
}
```

- The second example shows how to specify dependencies as well as defining versioning (i.e. compatibility keys)
- The package name imposes a directory structure. In the 2nd case, the video_copy package is located at:
  
  
  (repository_path)/servers/video_copy/

package.xs

```
function getLibs(prog)
{
  if (prog.build.target.isa == "64P") {
    if ( this.MYALG.Debug == false )
      lib = "lib/myalg_tto_le_release.a64P";
    else
      lib = "lib/myalg_tto_le_debug.a64P";
  }
  return (lib);
}

function getStackSize(prog)
{
  return (64 * 1024);
}
```

- While this is an .xs file for a codec package, it demonstrates how build-time methods can be defined for a package
  - The first function returns a library based on the value of a (.cfg) config option
  - The second tells the wizard (and/or Configuro) how big the stack size should be
- In most cases, we can leave it up to the GUI wizard tools to create this file for us
package.bld

```
var Pkg = xdc.useModule('xdc.bld.PackageContents');
/*
 * ======== Add Executable ========
*/
Pkg.addExecutable(
  "app_debug", // Name of executable file
  MVArm9, // Build target (from config.bld)
  MVArm9.platform, // Platform (from config.bld)
  {
    cfgScript: "app_cfg.cfg", // App/engine or server cfg file
    profile: "debug", // Build profile to be used
  }
).addObjects( ["main.c"] ); /* JavaScript array of obj files; if source files are passed, xdc will first build them into object files */
```

- The `addExecutable` method tells XDC what (and how) to build your executable program; there are many other methods, ex: `addLibrary`, `addOtherFiles`, `makeEpilog`
- This example is a "hard-coded" version and would need to be edited for program; the lab solutions contain a more generic version of package.bld

config.bld

```
Parts of config.bld:

- **DSP Target**
- **ARM Target**
- **Linux Host Target**
- **Build Targets**

// ======== Linux host target ========
var Linux86 = xdc.useModule('gnu.targets.Linux86');

// ======== Arm target ========
var MVArm9 = xdc.useModule('gnu.targets.MVArm9');

// ======== DSP target ========
var C64P = xdc.useModule('ti.targets.C64P');
C64P.platform = "ti.platforms.evmDM6446";
C64P.ccOpts.prefix += "-k al";

// location of your C6000 codegen tools
C64P.rootDir = java.lang.System.getenv("C6000_CG");

// ======== Build targets ========
// list of targets (ISAs + compilers) to build for Build.targets = [
MVArm9,
Linux86,
C64P
];
```
Lab 12

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.

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.
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 ~/workshop/lab12a_build_server/app
   cp -R -u ~/workshop/Lab11a_publish_server/app/* .
   ```

2. Change to the /home/user/workshop/lab12a_build_server/osdfiles and copy the files from your previous lab exercise.
   
   ```
   cd ~/workshop/lab12a_build_server/osdfiles
   cp -R -f ~/workshop/Lab11a_publish_server/osdfiles/* .
   ```

3. Change into the Lab12a_server 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 ~/workshop/lab12a_build_server/server
   gedit makefile &
   ```

   (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:
  
  ```shell
  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:

  ```shell
  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.

   ```shell
   make 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.evmDM6446 (to match our board)
   - **Package Name:** server (to match the name from our last lab)
   - **Output Repo:** /home/user/workshop/lab12a_build_server
     
     (This means the full path to our package will be: /home/user/workshop/lab12a_build_server/server)
   - **Codecs:** Check the algo’s we have been using in the last exercises:
     
     ```shell
     AUDDEC_COPY
     AUDENC_COPY
     VIDDEC_COPY
     VIDENC_COPY
     ```

6. Fill in the second screen of the DSP Server Wizard.

   Click Next

   We will use the default values for everything in the second step of the wizard except the algorithm names and video codecs groupid. We will change them to match the strings we have previously been using in our application.

   ```shell
   VIDDEC_COPY: video_decoder
   VIDENC_COPY: video_encoder
   AUDDEC_COPY: audio_decoder
   AUDENC_COPY: audio_encoder
   ```

   Put in Group 0 - to match server.cfg defaults for current version of the DSP Server Wizard
7. **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:
   
   ~/workshop/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.

---

**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)**

8. **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.

**server.cfg**

This is also a 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.

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. (Note, we will edit one or two of these files in Lab 13, though.)

Build the Server
9. Build the server package.
   Using the makefile again, run XDC to build the server.
   
   make 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.

10. Move back to the application directory.
   cd ../app
   to move back to the lab12a_build_server/app directory

11. Edit the makefile_profile.mak due to the name change of the server executable.
   
   gedit makefile_profile.mak &
   Edit the XDCPATH variable in to add the current lab’s directory by adding 
   /home/user/workshop/lab12a_build_server to beginning of the XDCPATH. (Note, you could try using a relative path once you get to work the first time.)
   Also, change the clean and install rules to reflect the new server.x64P name.

12. 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.

13. Execute the app_DEBUG.x470MV application. Press ctrl-c to exit the application.
Lab12b_real_h264

In this lab exercise, we will replace the simple copy-based video codecs that we have been using with a real H.264 encoder and decoder pair.

Install New Codecs and Inspect Them

14. Begin by installing the H.264 evaluation codecs.

We’ve placed a copy of the H.264 codecs into the VMware shared folder.

When evaluating codecs from TI or third parties, you will download an archive file. Generally you can install them wherever you want within the filesystem; for our exercises, though, we recommend you untar the files so that it creates a “watermarked” directory in your /home/user folder.

```bash
    cd ~
    tar zxfv /mnt/hgfs/shared/watermarked_h264.tar.gz
```

15. Inspect the package names.

After you untar the h.264 codecs, inspect the following two packages (h264enc, h264mpdec)

Examine the “package name” defined for each package (in package.xdc):

- h264enc/package.xdc
- h264mpdec/package.xdc

Notice that the encoder package name is `ti.sdo.codecs.h264enc` (which matches the package path) with a module named H264ENC; with similar results for the H264MPDEC. These package and module names will be needed in your .cfg file to import the codecs into your new DSP server (instead of using viddec_copy and videnc_copy used in earlier labs). The server wizard (in step 20) will create our new .cfg file for us.

You will also need to update your makefiles with the repository path for these new codecs; this will be completed in step 19 (server makefile) and 26 (application makefile_profile.mak).
File Management

16. Clean-up your lab12a_build_server/app directory to minimize files to copy.

   cd ~/workshop/lab12a_build_server/app
   make clean

17. Copy your lab12a_build_server lab forward into lab12b_real_h264 as a starting point.

   Copy the files from the osdfiles and app folders:

   cd ~/workshop/lab12b_real_h264/osdfiles
   cp -R ~/workshop/lab12a_build_server/osdfiles/* .

   cd ../app
   cp -R ~/workshop/lab12a_build_server/app/* .

   Since we’re going to use the server wizard again to create the necessary files, we’ll only need to copy over our previous makefile:

   cd ../server
   cp -R ~/workshop/lab12a_build_server/server/makefile .

   Note: If you didn’t get lab12a_build_server exercise to work, just copy the corresponding directory from the ~/solutions/lab12a_build_server directory instead.

Create/Build the Server (which now contains H.264 codecs)

18. Navigate to the server folder.

   You should still be in the /home/user/workshop/lab12b_real_h264/server directory. If not, please change to that directory

19. Modify XDCPATH in the server’s makefile to include the new codecs and server location.

   We need to add the repository location for our new H.264 video codecs, as well as the repository directory for the server we are building. You could just add these absolute directories to the XDCPATH statement, though we chose to do so thru make variables as shown here:

   LAB12a_DIR := /home/user/workshop/lab12a_build_server
   LAB12b_DIR := /home/user/workshop/lab12b_real_h264
   H264_DIR := /home/user/watermarked/packages
   LAB_DIR := $(LAB12b_DIR);$(H264_DIR)
   export XDCPATH := $(LAB_DIR);$(CONFIG_BLD_PATH);$(CE_INSTALL_DIR)/packa ...
20. Create the server configuration files for our new H.264 server.

Re-run the DSP Server Wizard similar to the steps from Lab12a (page 12-24). Kick off the wizard using

```
make run_server_wizard
```

Notes:

- As we did in lab12a, use the package name “server”.
- Make sure you choose the correct output repository:
  ```
  /home/user/workshop/lab12b_real_h264
  ```
- This time, choose the H264 video encoder and decoder codecs (instead of videc_copy and videnc_copy).
- In the wizard’s second dialog – make sure you re-name the codecs as:
  ```
  audio_decoder, audio_encoder, video_decoder, and video_encoder.
  ```
21. Edit the `codec.cfg` file created by the server wizard.

In `codec.cfg`, replace “undefined” and “DDR” with “DDR2”; also comment out 2 lines.

```c
var H264ENC = xdc.useModule('ti.sdo.codec.s264enc.ce.H264ENC

// Module Config
H264ENC.manageIn BufsCache = [true, true, true, true.
H264ENC.manageOutBufsCache = [true, true, true, true.
H264ENC.manageReconBufsCache = [true, true, true, true.
H264ENC.alg.watermark = true;
H264ENC.alg.codeSection = "DDR2";
H264ENC.alg.udataSection = "DDR2";
H264ENC.alg.dataSection = "DDR2";

var H264MPDEC = xdc.useModule('ti.sdocodec.s264mpdec.ce.H26

H264MPDEC.alg.dataSection = "DDR2";
H264MPDEC.alg.codeSection = "DDR2";
H264MPDEC.alg.IMX_IMAGEBUF = "DDR2";
H264MPDEC.alg.IMX_IMAGEBUF_1 = "DDR2";
H264MPDEC.alg.OVERLAY = "DDR2";
H264MPDEC.alg.useOverlay = true;
// H264MPDEC.alg.imbufCodeOverlaySymbol = undefined;
// H264MPDEC.alg.imbuf1CodeOverlaySymbol = undefined;
```

We also need to edit the `groupId`'s found in the array of server algorithms towards the bottom of the `config.cfg` file. Basically, we need these to match the ‘group’ setting in the `server.cfg` file.

For example:

```c
codec.cfg

{name: "video_encoder", mod: H264ENC,
  stackWemId: 0, priority: ServerConfig.HIGHEST_PRIORITY,
  groupId : 0,
},

{name: "video_decoder", mod: H264MPDEC,
  stackWemId: 0, priority: ServerConfig.HIGHEST_PRIORITY,
  groupId : 0,
},
```

We could just have easily changed the `groupId` settings in the `server.cfg` file; they really just need to match.

**Note:** Future versions of the Server Wizard do a better job matching these items for you, which should reduce your busy work.)
22. **Edit the DARAM and SARAM scratch memory sizes for groupId 0.**

The DSP Server Wizard takes a guess at what scratch memory sizes are required for algorithm scratch groups. Here is the default provided by the DSP Server Wizard:

```c
// Finally, note that if the codecs correctly implement the
// ti.sdo.cs.ICodec.getDaramScratchSize() and .getSaramScratchSize() methods,
// this scratch size configuration can be autogenerated by
// configuring Server.autoGenScratchSizeArrays = true.

DSKT2.SARAM_SCRATCH_SIZES[0] = 32*1024;
DSKT2.SARAM_SCRATCH_SIZES[0] = 64768;
DSKT2.DARAM_SCRATCH_SIZES[0] = 64768;
```

If you examine the H.264 algorithm’s datasheet (see optional lab on page 12-37), you will notice the algorithm requires 63.25K bytes of memory. Unfortunately, the default only specifies half of the required memory. Further, it doesn’t specify both DARAM and SARAM scratch sizes.

As opposed to setting the numbers as in the file, we can solve the problem by using the “auto” feature specified in the comments just above. Thus we can edit the file in either of the two following ways.

Provide specific memory values:

```c
//DSKT2.SARAM_SCRATCH_SIZES[0] = 32*1024;
DSKT2.SARAM_SCRATCH_SIZES[0] = 64768;
DSKT2.DARAM_SCRATCH_SIZES[0] = 64768;
```

Or:

Use the “autoGenScratchSize” method:

```c
//DSKT2.SARAM_SCRATCH_SIZES[0] = 32*1024;
Server.autoGenScratchSizeArrays = true;
```

If you aren’t sure whether your algorithm provides these sizes, you can verify it by looking in the module’s .xs file. In the case of our H.264 encoder, look for the functions `getDaramScratchSize()` and `getSaramScratchSize()` in the file:

```c
~/watermarked/packages/ti/sdo/codecs/h264enc/ce/H264ENC.xs
```

As described earlier, the .xs files define the package and module specific methods that can be used by the RTSC build tools.

**Note:** Later versions of the DSP Server Wizard do a better job pulling this information from the codecs and setting these values for you.
23. **Edit the `package.xs` file created by the server wizard.**

One final thing before we can build our server. Comment out the last section from the `package.xs` file:

![Code from package.xs](image)

It should not be necessary to comment this section out in the next version of the wizard, but it currently causes build problems.

**Note:** Once again we note that later versions of the DSP Server Wizard do a better job pulling this information from the codecs and setting these values for you.

24. **Build the server.**

```
make build_server
```
Examine and Modify the App files

Update make script for new package locations

25. Change to app folder.
   Note, if you didn’t clean the folder before copying the files to it, we recommend doing that after changing to the app directory.
   ```
   cd ../app
   ```

26. Open makefile_profile.mak and change the XDCPATH and install references.
   Similar to the server’s makefile (in step 19), change the XDCPATH variable to reference the new codecs and server locations.
   ```
   XDCROOT := $(XDC_INSTALL_DIR)
   CONFIGURE := $(XDCROOT)/xs xdc.tools.configuro
   LAB12a_DIR := /home/user/workshop/lab12a_build_server
   LAB12b_DIR := /home/user/workshop/lab12b_real_h264
   H264_DIR := /home/user/watermarked/packages
   LAB_DIR := $(H264_DIR);$(LAB12b_DIR)
   export XDCPATH := $(LAB_DIR);$(CONFIG_BLD_PATH);$(CE_INSTDIR)
   ```
   Also, change the install rule so that the correct server.x64P file gets copied to the target.
   ```
   .PHONY : install
   install : $(PROGNAME)_$(PROFILE).x470MV
   @echo
   @echo "0. ----- Installing $(PROGNAME)_$(PROFILE).x470MV"
   @echo "Execution Directory: $(EXEC_DIR)"
   $(AT) mkdir -p $(EXEC_DIR)
   $(AT) cp $(LAB12b_DIR)/server/server.x64P $(EXEC_DIR)
   $(AT) cp $(LAB12b_DIR)/server/server.x64P $(EXEC_DIR)
   @echo "Install (i.e. copy) has completed" ;
   ```

Copy and Examine New Setup Functions

27. Copy the updated encoder and decoder files into the /app directory.
   Copy the following files:
   - `video_decoder.c`
   - `video_decoder.h`
   - `video_encoder.c`
   - `video_encoder.h`
   from the `lab12b_real_h264` directory to the `lab12b_real_h264/app` directory (overwriting the previous versions copied from `lab12a_build_server`).
   ```
   cp ../video_* .
   ```
   These new files provide modifications to `video_encoder_setup()` and `video_decoder_setup()` functions. Previously, we were using copy-based dummy codecs that did not require an accurate configuration of static and dynamic parameters. Now, we do.
28. Examine `video_encoder_setup()` and `video_decoder_setup()` functions.

Examine `video_encoder_setup()` in `video_encoder.c` and `video_decoder_setup()` in `video_decoder.c`. The easiest way to do this is to use a graphical ‘diff’ tool. We’ve installed one called **Beyond Compare**. (It highlights in red, the differences between the files which makes it easy to see the new changes.)

```
bcompare video_encoder.c /home/user/workshop/lab12a_build_server/app/video_encoder.c &
```

`video_encoder_setup()` – Notice that the following parameters have been added:

- frame height
- frame width
- bit rate
- frame rate
- return value for minimum output buffer size

```
bcompare video_decoder.c /home/user/workshop/lab12a_build_server/app/video_decoder.c &
```

`video_decoder_setup()` – Note the following additions:

- frame width
- frame height
- return value for minimum buffer size of the input buffer

As you can see, the `video_encoder_setup()` and `video_decoder_setup()` functions simply use these new parameters to build the `params` (static parameters) structure to pass to the create function and `dynamicParams` (dynamic parameters) structure to pass to the XDC_SETPARAMS ioctl. Since this is fairly rote work, we have given you these functions instead of having you type them in.
Modify video_thread.c

While we provided new encoder/decoder setup functions, you’ll need to edit these function calls in your own video_thread.c file so that their invocation matches the new prototypes we just examined.

29. Open and edit video_thread.c.

Open app/video_thread.c and modify the video_encoder_setup() and video_decoder_setup() function calls to support the newly added arguments.

video_encoder_setup():

Prototype:

```c
int video_encoder_setup( Engine_Handle, char *encoderName, VIDENC_Handle,
    int frameWidth,
    int frameHeight,
    int frameRate,
    int bitRate,
    int *minOutputSizeByRef);
```

You should specify the additional arguments as follows:

- int frameWidth        D1_WIDTH
- int frameHeight       D1_HEIGHT
- int frameRate         30000
- int bitRate           4000000
- int *minOutputSizeByRef &encBufSize

video_decoder_setup():

Prototype:

```c
int video_decoder_setup( Engine_Handle, char *decoderName, VIDDEC_Handle,
    int frameWidth,
    int frameHeight,
    int *minInputSizeByRef);
```

You should specify the additional arguments as follows:

- int frameWidth        D1_WIDTH
- int frameHeight       D1_HEIGHT
- int *minInputSizeByRef &encBufSize

Note that video_decoder_setup() is going to overwrite the encBufSize variable as it was set by video_encoder_setup(). Normally you would use two different variables here, compare them, and use the larger value to allocate the intermediate buffer, thereby ensuring that the minimum sizes required by both the encoder and decoder are met.

In our case, we are simply going set encBufSize to captureSize and not worry about the minimum sizes for each codec.
Build, Install and Run

30. Try building the application.

   make debug

You should get an error since the decoder codec was built using an earlier OSAL version.

   The error statement provides a hint on how to turn this error into a warning:

   Adding the following line to your program configuration script:
   environment['xdc.cfg.check.fatal'] = 'false';

Add this statement to the end of your app_cfg.cfg file to eliminate this error:

   var Engine = xdc.useModule('ti.sdo.ce.Engine');
   var myEngine = Engine.createFromServer('encodeddecode', // Engine name (as ref
                                          '/server.x64', // path to server exe,
                                          'server'       // server package
                                      );
   environment['xdc.cfg.check.fatal'] = 'false';

31. Build and install the new application using makefile in the app directory.

   With this final addition to your .cfg file, we should be ready to build, install, and run the
   H.264 code.

32. In a DVEVM terminal, run app_DEBUG.x470MV.

   Don’t forget the loadmodules.sh script if necessary to load the CMEM and DSPLINK
   modules.
(Optional) View Codec Datasheet

Examine the codec datasheet

33. Examine the H.264 encoder and decoder datasheets.

To learn more about the required stack sizes for these codecs, you can open the datasheets located in the package ti.sdo.codecs.h264:

```
/home/user/watermarked/packages/ti/sdo/codecs/h264enc/docs
```

Only the ENCODE directory has a datasheet. This is an older version of the codec and the DECODE datasheet was not available at the time. However, you’ll be able to get a feel for what a codec datasheet looks like by looking at the encode.pdf.

You may use the Evince application (PDF viewer in RH5) to view pdf files. For instance:

```
cd /home/user/watermarked/packages/ti/sdo/codecs/h264enc/docs
evince H264_Encoder_DM6446_Datasheet.pdf
```

**Note:** If evince does not display the PDF correctly, you may want to copy the file into the /mnt/hgfs/shared directory and view it within Windows with Acrobat Reader.

Within the Encoder Datasheet, note Table 3 – Memory Statistics. The second-to-last column shows the stack memory requirement of “8” and the note at the bottom of the table indicates all memory is expressed in kilobytes (1 kilobyte = 1K = 1024 bytes). Thus, by reading this table, we would know the encoder requires 8192 bytes of stack.

If you refer back to the class discussion, you might remember there is a thread attribute (in the .cfg file) which describes the algorithms stack size.

. Did the server wizard specify the algorithms stack size? _________________________________

. If not, why not? ____________________________________________________________

A codec can specify the (default) stack size in it’s RTSC metadata. Examine the H264ENC.xs file to see how RTSC can provide the default stack size to the build system.

```
cd /home/user/watermarked/packages/ti/sdo/codecs/h264enc/ce
gedit H264ENC.xs &
```

Note, you can override this default by adding a configuration parameter in that codec’s **thread attributes** structure within **codec.cfg**.
This application will record video in an h.264 format. It also records audio, although, since we are using a copy-based audio codec, it actually records the audio in a raw format.

**Examine Files**

34. **Examine the contents of lab12c_h264_record.**

    Change to /home/user/workshop/lab12c_h264_record/app and examine the contents.

    The following source files have changed in this application:
    - video_thread.c
    - audio_thread.c

    The following source files are no longer needed by this application:
    - video_decoder.c
    - video_decoder.h
    - video_output.c
    - video_output.h
    - video_osd.c
    - video_osd.h

**Examine “Poor Man’s” A/V Sync.**

35. **Examine the time stamping used in audio_thread.c and video_thread.c.**

    Open audio_thread.c and video_thread.c.

    We have used what is called a “poor-man’s” version of syncing audio and video (A/V sync).

    The gettimeofday() function has been used to acquire a timestamp for each audio buffer and video frame. These timestamps are in a timeval structure which has both a seconds and a microseconds field. This timeval structure is then written into a “save” file before each audio or video buffer is used by the player application. These timestamps will be used to synchronize the audio and video streams.

    This synchronization is absolutely necessary in this application because the video stream is constantly dropping frames. The H.264 encoder executes at 15-20 frames per second for 720x480 frames on this device, but the video is coming in at 30 frames per second. If the player simply plays the frames out, not only will the video play out about 50% faster than it should, but it will quickly diverge from the audio. The time stamping added in here manages to at least keep the streams close to being aligned.

    This is very much a “poor man’s” audio-visual synchronization scheme. However, and as you may see in lab13c_h264_player (next lab), mis-synchronization is sometimes apparent in dialogue when it is played back. The reasons that this scheme is not particularly accurate are discussed at the end of this lab section. Nonetheless, this scheme does at least get the synchronization close (within about a tenth of a second, maintainable over any record length), and may even be sufficient for certain applications.
Build, Install and Run the Application

36. Build and install the recorder application.

```bash
cd ~/workshop/lab12c_h264_record  # (if you're not already there)
make debug install
```

37. Run the application.

On the DVEVM, move to the `/opt/workshop` directory.

Execute the `mount_harddrive.sh` script to mount the hard drive into the filesystem. (This should have been copied along with the `app` and `server`.)

Execute `app_release.x470MV` to begin recording audio and video to the hard drive. (Note, there won’t be anything showing on the screen, as it’s similar to the record program we ran back lab07b.)

Press ctrl-c to exit.

38. List the generated files.

```bash
ls -lsa /mnt/harddrive/opt/workshop
```

You should see that the file size of the compressed video file, `video.h264`, is only two or three times as large as the raw audio file, `audio.raw`. For NTSC video, this is significant compression. Let’s see how this compressed video looks in the next section.
(Optional) Lab12d_h264_playback

This application will playback h.264 encoded video from a file. It also plays back raw audio streams. We will use this application to play back the files recorded in lab13b_h264_record.

To give you a better view of the video, we have removed the on-screen displays, although there is plenty of horsepower left on the device to support this, and the challenge exercise, if you have time remaining, is to add your own custom OSD back into the playback.

Examine Files and Timestamping

39. Examine the files in the directory.

   Change to /home/user/workshop/lab12d_h264_playback/app and examine the contents.

   The following files have been modified from the previous lab:

   - video_thread.c
   - audio_thread.c

   The capture and encoding files have been removed because they are no longer needed.

40. Examine the time stamping.

   We have again used a poor man’s A/V sync in audio_thread.c and video_thread.c.

   The basic method of the player’s A/V sync is to set the Linux system clock to match the timestamps coming off of the audio stream as it is read. This is not particularly good practice, but is quick and simple and has the advantage that Linux will automatically update the system clock in between timestamp settings, providing a more-or-less continuous master clock for the video stream to run off of.

   The audio stream reads each audio timestamp from the file as it reads the corresponding buffer and uses settimeofday() to set the system clock with the timestamp. The video thread then reads the timestamp for each video frame from its video file when it reads the corresponding video frame and compares the video frame timestamp to the system timestamp, which is also the audio timestamp.

   In this application, because we are using H.264 which, at this resolution and frame rate, will always drop frames, we know that we do not need to adjust for the case that the video timestamp is earlier than the audio timestamp. The application should only ever need to pause the video frame so that the audio stream can catch up. In a real application, however, we should also test for the opposite case and drop a video frame if the video timestamps ever get more than about half of a frame ahead of the audio stream.
Build, Install and Run the Application.

41. Build, install and run the application.

Build and install the playback application.

On the DVEVM, move to the /opt/workshop directory.

Execute `app_release.x470MV` to play back the audio and video from the hard drive. Press `ctrl-c` to exit, or the application will exit when it reaches the end of the audio and video files.

You may notice that even with the time stamping measures we have put in place, the audio and video are slightly off, giving the dialogue a slightly overdubbed feel. We will discuss why this happens and how a more sophisticated solution could address the issues in the following discussion.

Discussion: A-V Sync

In our recorder and player, we have used a very simplistic method to synchronize the audio and video streams when we play them back from their recorded files. This scheme is good enough that it will generally synchronize the audio and video to within a tenth of a second, maintainable over any recording length. This is certainly an improvement over what would be seen if nothing was done for synchronization, but is not sufficient for many professional applications.

Why is this synchronization only accurate to about a tenth of a second when the `gettimeofday()` and `settimeofday()` functions that we’re using for our timestamps are accurate to within a microsecond? The answer comes from the fact that we are managing our timestamps from the application instead of from within the driver.

For instance, the player application uses the timestamp of each audio buffer to set the system clock using `settimeofday()` just before it writes the buffer to the driver. On first glance, this might seem to be the perfect solution. But recall that the audio driver is a buffered stream. The moment when the application writes an audio buffer to the audio driver is not the same moment that the audio buffer is output by the driver to the codec or DAC hardware to be played. Examination of `audio_thread_fxn()` will show that up to two buffers (of time period 3092 samples / 44K Hz = 70 mSec) may remain to be played before the first sample from this buffer can be output. A similar problem occurs from the buffer’s being read off of the incoming audio stream, which is similarly buffered.

Likewise, the timestamp that is taken when a video frame is dequeued from the v4l2 driver is not the time that the frame was sampled. Just the fact that the frame is being queued tells us that. Since we have three video frames, this error could put us off by as much as two frames, which also happens to be about 70 mSec.

It is by simple blind luck that in this application the 70 mSec that the video stream is off by and the 70 mSec that the audio stream is off by are similar timeframes and happen to compensate for each other instead of producing a 140 mSec offset. What happens is that the video timestamp is 70 mSec behind what it should be. As a result, the video frame is delayed by 70 mSec which tends to compensate for the 70 mSec delay in the buffered audio output. We still have the delay caused by the audio input, however, as well as general system jitter, including jitter due to switching between threads.
The result is that we have a system that tends to be off by 100 mSec, which would be much worse if the buffers in question were larger or if they didn’t happen to match fairly well. Certainly for a production system, we don’t want to have to count on the time-based errors in the system compensating for each other, and 100 mSec is probably not good enough for a professional system anyway.

What then is the solution? The answer is that it should be the driver and not the application that is managing the timestamp generation and compensation for the system. The audio and video input drivers should be time stamping the buffers when they are actually sampled. Likewise, the output drivers should be passed a timestamp indicating when the target time point for the corresponding buffer is to be played or displayed. The output drivers would then be responsible for measuring the target timestamp against the system clock and adjusting as necessary.

A fair output driver would simply compare the target timestamp to the system clock and either drop/repeat samples and frames as necessary to align the two. A good output driver would be smart enough to space its recovery over multiple samples or frames to reduce the jitter effect. An excellent output driver would be able to modify an on-board phase-locked loop to subtly adjust the clocking of the audio or video output hardware, therefore re-synchronizing the clocks without having to drop or repeat frames.

Unfortunately, a more detailed examination of these techniques is beyond the scope of this class, though audio-visual synchronization is an issue that system designers must be aware of in developing applications.
Authoring a xDAIS/xDM Algorithm

Introduction

This chapter looks at algorithms from the inside out; how you write a xDAIS algorithm. It begins with the general description of xDAIS and how it is used, then examines the interface standard by focusing on the creation/usage/deletion of an algorithm and how its API deals with memory resource allocations.

Learning Objectives

<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
</tr>
<tr>
<td>• What is xDAIS (and VISA)?</td>
</tr>
<tr>
<td>• Software Objects</td>
</tr>
<tr>
<td>• Master Thread Example</td>
</tr>
<tr>
<td>• Intro to Codec Engine Framework (i.e. VISA)</td>
</tr>
<tr>
<td><strong>Algorithm Lifecycle</strong></td>
</tr>
<tr>
<td><strong>Frameworks</strong></td>
</tr>
<tr>
<td><strong>Algorithm Classes</strong></td>
</tr>
<tr>
<td><strong>Making An Algorithm</strong></td>
</tr>
<tr>
<td><strong>Appendix</strong></td>
</tr>
</tbody>
</table>
# Chapter Topics

## Authoring a xDAIS/xDM Algorithm

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>13-3</td>
</tr>
<tr>
<td>What is xDAIS (and VISA)?</td>
<td>13-3</td>
</tr>
<tr>
<td>Software Objects</td>
<td>13-4</td>
</tr>
<tr>
<td>Master Thread Example</td>
<td>13-6</td>
</tr>
<tr>
<td>Intro to Codec Engine Framework (i.e. VISA)</td>
<td>13-6</td>
</tr>
<tr>
<td>Algorithm Lifecycle</td>
<td>13-9</td>
</tr>
<tr>
<td>Create</td>
<td>13-10</td>
</tr>
<tr>
<td>Process</td>
<td>13-14</td>
</tr>
<tr>
<td>Delete</td>
<td>13-16</td>
</tr>
<tr>
<td>Frameworks</td>
<td>13-17</td>
</tr>
<tr>
<td>Algorithm Classes</td>
<td>13-19</td>
</tr>
<tr>
<td>xDM/VISA Classes</td>
<td>13-19</td>
</tr>
<tr>
<td>Universal Class</td>
<td>13-20</td>
</tr>
<tr>
<td>Extending xDM</td>
<td>13-21</td>
</tr>
<tr>
<td>Design Your Own – Extending the Universal Class</td>
<td>13-23</td>
</tr>
<tr>
<td>Making an Algorithm</td>
<td>13-24</td>
</tr>
<tr>
<td>Rules of xDAIS</td>
<td>13-24</td>
</tr>
<tr>
<td>Using the Algorithm Wizard</td>
<td>13-26</td>
</tr>
<tr>
<td>Appendix</td>
<td>13-34</td>
</tr>
<tr>
<td>Reference Info</td>
<td>13-34</td>
</tr>
<tr>
<td>(Optional) xDAIS Data Structures</td>
<td>13-35</td>
</tr>
<tr>
<td>(Optional) Multi-Instance Ability</td>
<td>13-38</td>
</tr>
<tr>
<td>(Optional) xDAIS : Static vs Dynamic</td>
<td>13-40</td>
</tr>
</tbody>
</table>
Introduction

What is xDAIS (and VISA)?

**Algol’s as System Plug-In**

- Componentize algorithms for:
  - Plug-n-play **ease of use**
  - Single, **standardized interface** to use/learn
  - Enables use of common frameworks
- Express DSP Algorithm Interface Standard (xDAIS):
  - Similar to **C++ class** for algorithms
  - Provides a **time-tested**, real-time protocol

**Call with VISA : Author with xDAIS**

- Componentize algorithms for:
  - Plug-n-play **ease of use**
  - Single, **standardized interface** to use/learn
  - Enables use of common frameworks
- Express DSP Algorithm Interface Standard (xDAIS):
  - Similar to **C++ class** for algorithms
  - Provides a **time-tested**, real-time protocol
- Acronyms:
  - **xDAIS** – set of functions algorithm author writes (xDM – Extensions to xDAIS)
  - **VISA** – complimentary set of functions used by application programmer
Software Objects

Examples of software objects:
- C++ classes
- xDAIS (or xDM) algorithms

What does a software object contain?
- Thinking of C++ classes:
  Attributes:
  - Class object
  - Creation (i.e. construction) parameters
  Methods
  - Constructor
  - Destructor
  - “processing” method(s)

Comparing Objects: C++ / xDAIS

class algo{
public:
// methods
  int method1(int param);
  int method2(int param);
// attributes
  int attr1;
  int attr2;
}

typedef struct {
  // methods
  int (*method1) (int param);
  int (*method2) (int param);
  // attributes
  int attr1;
  int attr2;
} algo;

- xDAIS (and xDM) provide a C++-like object, implemented in C
- Because C does not support classes, structs are used
- Because structs do not support methods, function pointers are used
Comparing Methods: C++ / xDM

Create Instance: (C++ Constructor)
- C++ algo::algo(algo_params params)
- VISA VIDENC_create(VIDENC_params params)

Process:
- C++ algo::myMethod(...)
- VISA VIDENC_process(...)

Delete Instance: (C++ Destructor)
- C++ algo::~algo()
- VISA VIDENC_delete()

Note: With VISA, the framework (i.e. Codec Engine library) allocates resources on algorithm creation, as opposed to C++ constructors, which allocate their own resources.

Algorithm Creation

Traditionally algorithms have simply used resources without being granted them by a central source.

Benefits of Central Resource Manager:
1. Avoid resource conflict during system integration
2. Facilitates resource sharing (i.e. scratch memory or DMA) between algorithms
3. Consistent error handling when dynamic allocations have insufficient resources
Master Thread Example

Linux System

User Program

// "Master Thread"
// Create Phase
Initialize Drivers
Create Algo Instance
// Execute Phase
while(run)
  Input (exch bufs)
  Process
  Output (exch bufs)
// Delete Phase
Delete Algo Instance

VISA API
- create
- process
- control
- delete

Driver API
- open
- read
- write
- ioctl
- close

Linux Kernel

Intro to Codec Engine Framework (i.e. VISA)

Codec Engine Framework Benefits

- Multiple algorithm channels (instances)
- Dynamic (run-time) algorithm instantiation
- Plug-and-play for algorithms of the same class (inheritance)
- Sharing of memory and DMA channel resources
- Algorithm interoperability with any CE-based Framework
- Same API, no new learning curve for all algorithm users
- Provided by TI!

Many of these benefits are a direct result of the object-oriented structure of the codec engine
VISA API (Application Programming Interface)

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  - `create`
  - `delete`
  - `process`
  - `control`

- **Create**: creates an instance of an algo that is, it malloc's the required memory and initializes the algorithm
- **Process**: invokes the algorithm calls the algorithms processing function passing descriptors for in and out buffers
- **Control**: used to change algo settings algorithm developers can provide user controllable parameters
- **Delete**: deletes an instance of an algo opposite of “create”, this deletes the memory set aside for a specific instance of an algorithm

VISA – Eleven Classes

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  - `create`
  - `delete`
  - `process`
  - `control`

- VISA = 4 processing domains:
  - Video
  - Imaging
  - Speech
  - Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  - VISA Encoders/Decoders
  - Video ANALYTICS & TRANSCODE

  Universal (generic algorithm i/f)
  - New!

- V
  - VIDENC
  - VIDDEC
- I
  - IMGENC
  - IMGDEC
- S
  - SPHENC
  - SPHDEC
- A
  - AUDENC
  - AUDDEC

- Other
  - VIDANALYTICS
  - VIDTRANSCODE
  - Universal

Codec Engine : VISA API

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  - `create`
  - `delete`
  - `process`
  - `control`

- VISA = 4 processing domains:
  - Video
  - Imaging
  - Speech
  - Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  - VISA Encoders/Decoders
  - Video ANALYTICS & TRANSCODE

  Universal (generic algorithm i/f)
  - New!

- TI’s CODEC engine (CE) provides abstraction between VISA and algorithms
- Application programmers can purchase xDM algorithms from TI third party vendors
  - or, hire them to create complete SPL soln’s
- Alternatively, experienced DSP programmers can create xDM compliant algs (discussed next)
- Author your own algos or purchase depending on your DSP needs and skills

Reducing dozens of functions to 4

Complexity

DaVinci/OMAP System Design Workshop - Authoring a xDAIS/xDM Algorithm
Master Thread Key Activities

```c
idevfd = open("/dev/xxx", O_RDONLY);
ofilefd = open("./fname", O_WRONLY);
ioctl(idevfd, CMD, &args);
myCE = Engine_open("vcr", myCEAttrs);
myVE = VIDENC_create(myCE, "videnc", params);
while( doRecordVideo == 1 ) {
    read(idevfd, &rd, sizeof(rd));
    VIDENC_process(myVE, …);
    VIDENC_control(myVE, …);
    write(ofilefd, &wd, sizeof(wd));
}
close(idevfd);
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

Note: the above pseudo-code does not show double buffering, often essential in Realtime systems!

VISA Function Details

```c
Engine_Handle myCE;
AUDENC_Handle myAE;
AUDENC_Params params;
AUDENC_DynParams dynParams;
AUDENC_Status status;
CERuntime_init();
myCE = Engine_open("myEngine", NULL);
myAE = AUDENC_create (myCE, "aEncoder", &params);
stat = AUDENC_process(myAE, &inBuf, &OutBuf, &inArgs, &outArgs);
stat = AUDENC_control(myAE, XDM_GETSTATUS, &dynParams, &status);
AUDENC_delete(myAE);
Engine_close (myCE);
```

- **Engine** and **Codec** string names are declared during the **engine config** file
- The config file (.cfg) specifies which algorithm packages (i.e. libraries) should be built into your application

Pick your algo’s using .CFG file

```c
/* Specify your operating system (OS abstraction layer) */
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.LINUX;

/* Specify which algo’s you want to build into your program */
var vidDec = xdc.useModule('ti.codecs.video.VIDENC');
var audDec = xdc.useModule('ti.codecs.audio.AUDENC');

/* Add the Codec Engine library module to your program */
var Engine = xdc.useModule('ti.sdo.ce.Engine');

/* Create engine named “myEngine” and add these algo’s to it */
var myEng = Engine.create("myEngine",
    [  
        {name: "vEncoder", mod: vidDec, local: true},  
        {name: "aEncoder", mod: audDec, local: true},  
    ]);  
```
## Algorithm Lifecycle

### Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create (&quot;Constructor&quot;)</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>doFilter</td>
</tr>
<tr>
<td>Delete (&quot;Destructor&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

- Codec Engine only uses the Dynamic features of xDAIS

### Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Dynamic</th>
</tr>
</thead>
</table>
| Create ("Constructor") | algNumAlloc  
                          algAlloc  
                          algInit  |
| Process             | doFilter|
| Delete ("Destructor") | algFree |

- Codec Engine only uses the Dynamic features of xDAIS
iAlg Functions Summary

◆ Create Functions
  • `algNumAlloc` - Tells application (i.e. CODEC engine) how many blocks of memory are required; it usually just returns a number
  • `algAlloc` - Describes properties of each required block of memory (size, alignment, location, scratch/persistent)
  • `algInit` - Algorithm is initialized with specified parameters and memory

◆ Execute Functions
  • `algActivate` - Prepare scratch memory for use; called prior to using algorithms process function (e.g. prep history for filter algo)
  • `algDeactivate` - Store scratch data to persistent memory subsequent to algo’s process function
  • `algMoved` - Used if application relocates an algorithm’s memory

◆ Delete Function
  • `algFree` - Algorithm returns descriptions of memory blocks it was given, so that the application can free them

Create

Instance Creation - start

```
1. Here’s the way I want you to perform…
   `Params = malloc(x);
   *Params = PARAMS;
```

Diagram:
```
   Application Framework
                     +--- Params +--- Algorithm
    Application
   Framework
```

DaVinci/OMAP System Design Workshop - Authoring a xDAIS/xDM Algorithm
Algorithm Parameters (Params)

- How can you adapt an algorithm to meet your needs?

  Vendor specifies "params" structure to allow user to set creation parameters.
  These are commonly used by the algorithm to specify resource needs and/or they are used for initialization.

- For example, what parameters might you need for a FIR filter?

A filter called IFIR might have:

```c
typedef struct IFIR_Params {
    Int size;       // size of params
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
} IFIR_Params;
```

Instance Creation - start

```
Application
Framework

1. Here’s the way I want you to perform…
Params = malloc(x);
*Params = PARAMS;

2. How many blocks of memory will you need to do this for me?

3. I’ll need “N” blocks of memory.
   (N may be based upon a params value)

4. I’ll make a place where you can tell me about your memory needs…
MemTab = malloc(5*N)
```

```
Algorithm

```

```c
algNumAlloc() -->
N -->
MemTab
```
XDAIS Components: Memory Table

- What prevents an algorithm from “taking” too much (critical) memory?
  - Algorithms cannot allocate memory.
  - Each block of memory required by algorithm is detailed in a Memory Table (memtab), then allocated by the Application.

- MemTab:

<table>
<thead>
<tr>
<th>MemTab</th>
<th>Space: Internal / External memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Attributes: Scratch or Persistent memory (discussed later)</td>
</tr>
<tr>
<td>Alignment</td>
<td>Base Addr</td>
</tr>
<tr>
<td>Space</td>
<td>Base: Starting address for block of memory</td>
</tr>
</tbody>
</table>

MemTab example:

- Application
  - Based on the four memory details in MemTab,
  - Application allocates each memory block, and then
  - Provides base address to MemTab

- Algorithm
  - Algo provides info for each block of memory it needs,
  - Except base address …
Algorithm Lifecycle

**Instance Creation - finish**

5. Tell me about your memory requirements…

7. I’ll go get/assign the memory you need…
   
   ```c
   for(i=0;i<=N;i++)
   mem = malloc(size);
   ```

8. Prepare the new instance to run!

10. Delete MemTab

6. I’ll enter my needs for each of the N blocks of memory, given these parameters, into the MemTab…

9. Initialize vars in my instance object using Params & Base’s

**Example: Params & InstObj**

1. **Creation Params**

   ```c
   typedef struct IFIR_Params {
     XDAS_Uint32 size;
     XDAS_Int16 firLen;
     XDAS_Int16 blockSize;
   } IFIR_Params;
   ```

2. **memTab**

   ```c
   typedef struct IFIR_Obj {
     IFIR_Fxns *fxns;
     XDAS_Int16 firLen;
     XDAS_Int16 blockSize;
     XDAS_Int16 *blockPtr;
     XDAS_Int16 *historyPtr;
     type myGlobVar1;
     type myGlobVar2;
     type etc …
   } IFIR_Obj;
   ```
Algorithm Lifecycle

Process

Instance Execution

1. Get ready to run. Scratch memory is yours now.

\[ \text{algActivate()} \]

2. Prepare scratch memory, as required, from persistent memory

Scratch vs Persistent Memory

- **Scratch**: used by algorithm during execution only
- **Persistent**: used to store state information during instance lifespan

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scratch</th>
<th>Per.A</th>
<th>Per.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>Scratch A</td>
<td></td>
<td>Per.A</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>Scratch B</td>
<td>Per.B</td>
<td></td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
<td>Scratch B</td>
</tr>
</tbody>
</table>

Okay for speed-optimized systems, but not where memory efficiency is a priority ...

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scratch</th>
<th>Per.A</th>
<th>Per.B</th>
<th>Per.C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>Scratch A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm B</td>
<td>Scratch B</td>
<td>Per.B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm C</td>
<td>Scratch C</td>
<td></td>
<td>Per.B</td>
<td>Per.C</td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
<td>Per.B</td>
<td>Per.C</td>
</tr>
</tbody>
</table>

Usually a Limited Resource (e.g. Internal RAM)  Often an Abundant Resource (e.g. External RAM)
Example of Benefit of Scratch Memory

Example:
- Let's say we will process 1K block of data at a time
- For 32-tap filter, 32 samples must be saved from one process call to the next

<table>
<thead>
<tr>
<th># Chans</th>
<th>No Overlay / Scratch</th>
<th>Use Scratch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>1032</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>1064</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10,000</td>
<td>1320</td>
</tr>
</tbody>
</table>

Without using scratch (i.e. overlay) memory, 10 channels of our block filter would take ten times the memory.

If sharing the block between channels, memory usage drops considerably. Only 32 RAM/channel persistent buffer to hold history vs. 1000 RAM/channel.

Instance Execution

Application Framework

1. Get ready to run. Scratch memory is yours now.

2. Prepare scratch memory, as required, from persistent memory

3. Run the algorithm ...

4. Perform algorithm - freely using all memory resources assigned to algo

5. I need the scratch block back from you now...

6. Save scratch elements to persistent memory as desired

Algorithm

--- algActivate() ---

--- process() ---

--- algDeactivate() ---
Delete

Instance Deletion

Application Framework

If I no longer need the algorithm:
1. I’ll make a memTab again, or reuse the prior one
   *memTab = malloc(5*N)

2. What memory resources were you assigned?

4. free all persistent memories recovered from algorithm

Algorithm

InstObj

Param1
Param2...
Base1
Base2...

MemTab

size
alignment
space
attrs
*base

algFree()
If all algorithms must use these ‘create’ functions, couldn’t we simplify our application code?

One create function can instantiate any XDM algo

History of algorithm creation functions from TI:
- ALG create is simplistic example create function provided with the xDAIS library
- ALGRE library provided in Reference Frameworks
- DSKT2 library is used by the Codec Engine and Bridge Frameworks
- Codec Engine (CE) provides create functions defined in xDM (or xDM-like) algs

Create Functions

Codec Engine Framework (VISA)

- algNumAlloc ()
- algAlloc ()
- algInit ()

◆ These functions are common for all xDAIS/xDM algo’s
VISA – CODEC Engine - xDM

VISA API Layer: Application Programmer

VIDDEC_create() VIDDEC_control() VIDDEC_process() VIDDEC_delete()

CODEC Engine framework: TI

algNumAlloc algAlloc MEM_alloc algInit
control algActivate process algDeactivate
algNumAlloc algFree MEM_free

xDM Algorithm: DSP Algo Author

algNumAlloc ---------------------
algAlloc ---------------------
alginitial ---------------------
algFree ---------------------
algActivate ---------------------
algDeactivate ---------------------
process ---------------------
control ---------------------

VISA – CODEC Engine - xDM

VISA API Layer: Application Programmer

VIDDEC_create() VIDDEC_control() VIDDEC_process() VIDDEC_delete()

CODEC engine is provided by TI
You need only be concerned with VISA or xDM

xDM Algorithm: DSP Algo Author

algNumAlloc ---------------------
alginitial ---------------------
algFree ---------------------
algActivate ---------------------
algDeactivate ---------------------
process ---------------------
control ---------------------
Algorithm Classes

xDAIS Limitations

- xDAIS defines methods for managing algo heap memory: algCreate, algDelete, algMoved
- xDAIS also defines methods for preparation/preservation of scratch memory: algActivate, algDeactivate
- Does not define the API, args, return type of the processing method
- Does not define the commands or structures of the control method
- Does not define creation or control structures

- Reason: xDAIS did not want to stifle options of algo author
- and ☹ Yields unlimited number of potential algo interfaces
- For DaVinci technology, defining the API for key media types would greatly improve
  - Usability
  - Modifiability
  - System design
- As such, the digital media extensions for xDAIS “xDAIS-DM” or “xDM” has been created to address the above concerns in DaVinci technology
- Reduces unlimited possibilities to 4 encoder/decoder sets!

xDM/VISA Classes

**Eleven xDM Classes**

- **Video**
  - VIDENC (encode)
  - VIDDEC (decode)
  - VIDANALYTICS (analysis)
  - VIDTRANSCODE (transcode)

- **Imaging**
  - IMGENC (encode)
  - IMGDEC (decode)

- **Speech**
  - SPHENC (encode)
  - SPHDEC (decode)

- **Audio**
  - AUDENC (encode)
  - AUDDEC (decode)

- **Universal** (custom algorithm)

- Create your own VISA compliant algorithm by inheriting the Universal class
- Then, use your algorithm with the Codec Engine, just like any other xDM algo
**Frameworks**

**xDM Benefits**

- Enable **plug + play** ability for multimedia codecs across implementations/vendors/systems
- **Uniform** across domains...video, imaging, audio, speech
- Flexibility - allows extension for custom / vendor-specific functionality
- **Low overhead**
- Insulate application from component-level changes
  - Hardware changes should not impact software (EDMA2.0 to 3.0,...)
  - PnP ...enable ease of replacement for versions, vendors
- **Framework Agnostic**
  - Integrate component into any framework
- Enable early and parallel development by **publishing** the API: create code faster
  - System level development in parallel to component level development
  - Reduce integration time for system developers
- **Published and Stable API**
  - TI, 3rd Parties, and Customers
  - Support Backward Compatibility

---

**Universal Class**

**Universal Algorithm : Methods**

```c
UNIVERSAL_create ( myCE, “aEncoder”, &IUNIVERSAL_Params );
UNIVERSAL_process ( IUNIVERSAL_Handle handle,
    XDM_BufDesc *inBufs,
    XDM_BufDesc *outBufs,
    XDM_BufDesc *inOutBufs,
    IUNIVERSAL_InArgs *inArgs,
    IUNIVERSAL_OutArgs *outArgs );
UNIVERSAL_control ( IUNIVERSAL_Handle handle,
    IUNIVERSAL_Cmd id,
    IUNIVERSAL_DynamicParams *params,
    IUNIVERSAL_Status *status );
UNIVERSAL_delete ( IUNIVERSAL_Handle handle );
```

- Create each of the required data structures:
  - `Params`, `InArgs`, `OutArgs`, `DynParams`, `Status`
- Structure names must begin with “I” and “my algo’s name”
  (which you can read as “interface” to “my algorithm”)
- Your algo’s structures must inherit the `IUNIVERSAL` datatypes
Universal Algorithm : Data

typedef struct IUNIVERSAL_Params {
    XDAS_Int32 size;
} IUNIVERSAL_Params;

//==============================================
typedef struct IUNIVERSAL_OutArgs {
    XDAS_Int32 size;
    XDAS_Int32 extendedError;
} IUNIVERSAL_OutArgs;

◆ Universal interface defined in xDM part of xDAIS spec
   /xdais_6_23/packages/ti/xdais/dm/iuniversal.h
   Which inherits:
   /xdais_6_23/packages/ti/xdais/dm/xdm.h
   Which then inherits:
   /xdais_6_23/packages/ti/xdais/ialg.h

Extending xDM

Easily Switch xDM Components

◆ All audio class decoders (eg: MP3 & AAC) provide the identical API
◆ Plug and Play: App using the IAUDDEC_Structures can call all audio decoders
◆ Any algorithm specific arguments must be set to default values internally by
  the vendor (insulating the application from need to specify these parameters)
◆ Specific functionality can be invoked by the app using extended data structures
◆ To summarize:
  • Most authors can use the default settings of the extended features provided by vendors
  • “Power users” can (optionally) obtain further tuning via an algos extended structures
Extending xDM – AAC DynamicParams Ex.

typedef struct IAUDDEC_DynamicParams {
    XDAS_Int32 size;       /* size of this structure */
    XDAS_Int32 outputFormat; /* To set interleaved/Block format */
} IAUDDEC_DynamicParams;

typedef struct IAACDEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
    Int DownSampleSbr;
} IAACDEC_DynamicParams;

AAC Control function code – Using the extended structure

XDAT_Int32 AACDEC_TII_control(IAUDDEC_Handle AAChandle, IAUDDEC_Cmd id, 
IAUDDEC_DynamicParams *params, IAUDDEC_Status *sPtr)
{
    IAACDEC_DynamicParams *dyparams = (IAACDEC_DynamicParams *)params;
    ...
    case IAACDEC_SETPARAMS:
        if(sizeof(IAACDEC_DynamicParams)==dyparams->auddec_dynamicparams.size)
            handle->downsamplerSBR=dyparams->DownSampleSbr;
        else
            handle->downsamplerSBR=0;

AAC and MP3 Extended Data Structures

typedef struct IAACDEC_Params {
    IAUDDEC_Params auddec_params;
} IAACDEC_Params;

typedef struct IMP3DEC_Params {
    IAUDDEC_Params auddec_params;
} IMP3DEC_Params;

typedef struct IAACDEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
    Int DownSampleSbr;    // AAC specific
} IAACDEC_DynamicParams;

typedef struct IMP3DEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
} IMP3DEC_DynamicParams;

typedef struct IAACDEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
} IAACDEC_InArgs;

typedef struct IMP3DEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
    XDAS_Int32   offset;
} IMP3DEC_InArgs;

typedef struct IAACDEC_OutArgs{
    IAUDDEC_OutArgs auddec_outArgs;
} IAACDEC_OutArgs;

typedef struct IMP3DEC_OutArgs{
    IAUDDEC_OutArgs auddec_outArgs;
    XDAS_Int32   layer;    // MP3 specific layer info
    XDAS_Int32   crcErrCnt;
}IMP3DEC_OutArgs;
Design Your Own – Extending the Universal Class

Creating a Universal Algorithm : Data

typedef struct IMYALG_Params {
    IUNIVERSAL_Params base;            // IUNIVERSAL_Params.size
    XDAS_Int32        param1;
    XDAS_Int32        param2;
} IMYALG_Params;

//===============================================
typedef struct IMYALG_OutArgs {
    IUNIVERSAL_OutArgs base;          // IUNIVERSAL_OutArgs.size
    // IUNIVERSAL_OutArgs.extendedError
    XDAS_Int32         outArgs1;
} IMYALG_OutArgs;

◆ Create each of the required data structures:
Params, InArgs, OutArgs, DynParams, Status
◆ Structure names must begin with “I” and “my algo’s name”
(which you can read as “interface” to ”my algorithm”)
◆ Your algo’s structures must inherit the IUNIVERSAL datatypes

Creating a Universal Algorithm : Methods

FIR_create ( myCE, “aEncoder”, &IFIR_Params );
FIR_process ( IUNIVERSAL_Handle handle,
               XDM_BufDesc *inBufs,
               XDM_BufDesc *outBufs,
               XDM_BufDesc *inOutBufs,
               IFIR_InArgs *inArgs,
               IFIR_OutArgs *outArgs );
FIR_control ( IUNIVERSAL_Handle handle,
              IFIR_Cmd id,
              IFIR_DynamicParams *params,
              IFIR_Status *status );
FIR_delete ( IUNIVERSAL_Handle handle );
Making an Algorithm

Rules of xDAIS

Application / Component Advantages

Dividing software between components and system integration provides optimal reuse partitioning, allowing:

- System Integrator (SI): to have full control of system resources
- Algorithm Author: to write components that can be used in any kind of system

What are “system resources”?

- CPU Cycles
- RAM (internal, external) : Data Space
- DMA hardware
  - Physical channels
  - PaRAMs
  - TCCs

How does the system integrator manage the usage of these resources?

Resource Management : CPU Loading

- All xDAIS algorithms run only when called, so no cycles are taken by algos without being first called by SI (application) code
- Algos do not define their own priority, thus SI’s can give each algo any priority desired – usually by calling it from a BIOS task (TSK)
- xDAIS algos are required to publish their cycle loading in their documentation, so SI’s know the load to expect from them
- Algo documentation also must define the worst case latency the algo might impose on the system
**Resource Management : RAM Allocation**

- **Algos never ‘take’ memory directly**
  - Algos tell system its needs (algNumAlloc(), algAlloc() )
  - SI determines what memory to give/lend to algo (MEM_alloc() )
  - SI tells algo what memories it may use (algInit() )

- **Algos may request internal or external RAM, but must function with either**
  - Allows SI more control of system resources
  - SI should note algo cycle performance can/will be affected

- **Algo authors can request memory as ‘scratch’ or ‘persistent’**
  - **Persistent**: ownership of resource must persist during life of algo
  - **Scratch**: ownership or resource required only when algo is running

**Resource Management : Scratch Memory**

- **SI can assign a permanent resource to a Scratch request**
  - Easy - requires no management of sharing of temporary/scratch resources
  - Requires more memory in total to satisfy numerous concurrent algos

- **SI must assure that each scratch is only lent to one algo at a time** (algActivate(), algDeactivate() )

- **No preemption amongst algos sharing a common scratch is permitted**
  - Best: share scratch only between equal priority threads – preemption is implicitly impossible
  - Tip: limit number of thread priorities used to save on number of scratch pools required
  - Other scratch sharing methods possible, but this is method used by C/E

- **Scratch management can yield great benefits**
  - More usage of highly prized internal RAM
  - Smaller total RAM budget
  - Reduced cost, size, and power when less RAM is specified
Using the Algorithm Wizard

Creating alg Interfaces: Component Wizard

This chapter documents the older wizard available in xDAIS 4.0. Find updated Algorithm Wizard available for purchase from NEXTXEN at: http://www.xdaiswizard.com

Information About the Component

![Image of component wizard interface with example screen capture]
Defining Parameters

typedef struct IAUDDEC_Params {
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 maxSampleRate; // max sampling frequency supported in Hz
    XDAS_Int32 maxBitrate; // max bit-rate supported in bits per sec
    XDAS_Int32 maxNoOfCh; // max number of channels supported
    XDAS_Int32 dataEndianness; // endianness of input data
} IAUDDEC_Params;

xDM : “Pre-defined” Params

IALG Functions – Common across all codec types

typedef struct IAUDDEC_Params { // structure used to initialize the algorithm
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 maxSampleRate; // max sampling frequency supported in Hz
    XDAS_Int32 maxBitrate; // max bit-rate supported in bits per sec
    XDAS_Int32 maxNoOfCh; // max number of channels supported
    XDAS_Int32 dataEndianness; // endianness of input data
} IAUDDEC_Params;
Define Additional Memory Blocks

- Define DSP Algorithm Function
  - xDM algs use the _process() function.
  - If using xDM, just enter its fully predefined function prototype into this dialog (along with the _control function).
Define Algorithm Parameters... Done

The component wizard has a dialog for creating the function’s arguments.

Since CW hasn’t been updated lately, xDM datatypes aren’t in the dropdown boxes – you’ll enter these manually.

Example Audio Decoder Data Structures

typedef struct XDM_BufDesc {
  XDAS_Int32 numBufs;
  XDAS_Int32 *bufSizes;
  XDAS_Int8 **bufs;
} XDM_BufDesc;

typedef struct IAUDDEC_InArgs {
  XDAS_Int32 size;
  XDAS_Int32 numBytes;
} IAUDDEC_InArgs;

typedef struct IAUDDEC_OutArgs {
  XDAS_Int32 size;
  XDAS_Int32 extendedError;
  XDAS_Int32 bytesConsumed;
} IAUDDEC_OutArgs;

// for buffer description (input and output buffers)
// number of buffers
// array of sizes of each buffer in 8-bit bytes
// pointer to vector containing buffer addresses

// for passing the input parameters for every decoder call
// size of this structure
// size of input data (in bytes) to be processed

// relays output status of the decoder after decoding
// size of this structure
// Extended Error code. (see XDM_ErrorBit)
// Number of bytes consumed during process call
Ready to “Generate Code”

Final Step…
Click on the
“Generate Code”
button!

View Code Written by Component Wizard

Reminder:
Now that you have generated your component, you need to perform the following steps:
1. Load the project in Code Composer Studio and add your specific algorithm code.
2. Run the DualIT program to verify your component is compliant.
3. Submit your algorithm for testing.

FILE NAME: FIR_TI_int.o
ALGORITHM: FIR
VERSION: T1
TARGET DSP: C55x
PURPOSE: Implementation of the FIR TI interface; TI’s implementation of the SPO interface.
xPressDSP Component Wizard Version 1.18.03 Auto-Generated Component
Number of Inputs: 0
Number of Outputs: 0
Creation Date: Mon - 29 October 2001
Creation Time: 02:44 AM
Component Wizard Made Instance Object

```c
/*
//=============================================
// FIR_TI_Obj
/*
typedef struct FIR_TI_Obj {
  IALG_Obj   alg;   /* MUST be first field of all FIR objs */
  XDAS_Int16 firLen;
  XDAS_Int16 blockSize;
  XDAS_Int16 * coeffPtr;
  XDAS_Int16 * workBuffer;
  XDAS_Int16 * historyBuffer;

  /* TODO: add custom fields here */
} FIR_TI_Obj;
```

Component Wizard Made algAlloc()

```c
Int FIR_TI_alloc(const IALG_Params *FIRParams, IALG_Fxns **fxns, IALG_MemRec memTab[])
{
  const IFIR_Params *params = (Void *)FIRParams;
  if (params == NULL) {
    params = &IFIR_PARAMS;  /* set default parameters */
  }

  memTab[0].size = sizeof(FIR_TI_Obj);
  memTab[0].alignment = (4 * 8) / CHAR_BIT;
  memTab[0].space = IALG_SARAM0;
  memTab[0].attrs = IALG_PERSIST;

  memTab[WORKBUFFER].size = (params->firLen+params->blockSize-1) * sizeof(XDAS_Int16);
  memTab[WORKBUFFER].alignment = (2 * 8) / CHAR_BIT;
  memTab[WORKBUFFER].space = IALG_SARAM0;
  memTab[WORKBUFFER].attrs = IALG_SCRATCH;

  memTab[HISTORYBUFFER].size = (params->firLen-1) * sizeof(XDAS_Int16);
  memTab[HISTORYBUFFER].alignment = (2 * 8) / CHAR_BIT;
  memTab[HISTORYBUFFER].space = IALG_EXTERNAL;
  memTab[HISTORYBUFFER].attrs = IALG_PERSIST;

  return (MTAB_NRECS);
}
```
Component Wizard Made `algFree()`

```c
Int FIR_TI_free(IALG_Handle handle, IALG_MemRec memTab[]) {
    Int n;
    FIR_TI_Obj *FIR = (Void *)handle;
    n = FIR_TI_alloc(NULL, NULL, memTab);

    memTab[WORKBUFFER].base = FIR->workBuffer;
    memTab[WORKBUFFER].size = (FIR->firLen+FIR->blockSize-1) * sizeof(XDAS_Int16);
    memTab[HISTORYBUFFER].base = FIR->historyBuffer;
    memTab[HISTORYBUFFER].size = (FIR->firLen-1) * sizeof(XDAS_Int16);
    return (n);
}
```

Component Wizard Made `algInit()`

```c
Int FIR_TI_initObj(IALG_Handle handle, const IALG_MemRec memTab[], IALG_Handle p, const IALG_Params *FIRParams) {
    FIR_TI_Obj *FIR = (Void *)handle;
    const IFIR_Params *params = (Void *)FIRParams;
    if(params == NULL){
        params = &IFIR_PARAMS; /* set default parameters */
    }

    FIR->firLen = params->firLen;
    FIR->blockSize = params->blockSize;
    FIR->coeffPtr = params->coeffPtr;
    FIR->workBuffer = memTab[WORKBUFFER].base;
    FIR->historyBuffer = memTab[HISTORYBUFFER].base;

    /* TODO: Implement any additional algInit desired */
    return (IALG_EOK);
}
```
algActivate & algDeactivate Incomplete…

Void FIR_TI_activate(IALG_Handle handle)
{
    FIR_TI_Obj *FIR = (Void *)handle;

    // TODO: implement algActivate
    // TODO: Initialize any important scratch memory values to FIR->workBuffer
}

Void FIR_TI_deactivate(IALG_Handle handle)
{
    FIR_TI_Obj *FIR = (Void *)handle;

    // TODO: implement algDeactivate
    // TODO: Save any important scratch memory values from FIR->workBuffer
    // to persistent memory.
}

algActivate / algDeactivate Completed

Void FIR_TI_activate(IALG_Handle handle)
{
    FIR_TI_Obj *FIR = (Void *)handle;

    memcpy((Void *)FIR->workBuffer, (Void *)FIR->historyBuffer,
           (FIR->firLen-1) * sizeof(Short));
}

Void FIR_TI_deactivate(IALG_Handle handle)
{
    FIR_TI_Obj *FIR = (Void *)handle;

    memcpy((Void *)FIR->historyBuffer, (Void *)FIR->workBuffer +
            FIR->blockSize, (FIR->firLen-1) * sizeof(Short));
}
Appendix

Reference Info

References

- Codec Engine Algorithm Creator User's Guide
  SPRUED6 Texas Instruments
- Codec Engine Server Integrator's Guide
  SPRUED5 Texas Instruments
- xdttools_1_21/doc directory in DVEVM 1.1 software
  Documentation on XDC Configuration Kit and BOM
- Using adapters to run xDAIS algorithms in the Codec Engine
  SPRAAE7 Texas Instruments

Glossary

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codec Engine</td>
<td>DaVinci framework for instantiating and using remote or local codecs</td>
</tr>
<tr>
<td>DMAN</td>
<td>Dma MANager module. Manages DMA resource allocation</td>
</tr>
<tr>
<td>DSKT2</td>
<td>Dsp SocKeT module, rev. 2. Manages DSP memory allocation</td>
</tr>
<tr>
<td>DSP Link</td>
<td>Physical Transport Layer for Inter-processor Communication</td>
</tr>
<tr>
<td>Engine</td>
<td>CE framework layer for managing local and remote function calls</td>
</tr>
<tr>
<td>EPSI API</td>
<td>Easy Peripheral Software Interface API. Interface to system drivers.</td>
</tr>
<tr>
<td>OSAL</td>
<td>Operating System Abstraction Layer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>Server</td>
<td>Remote Thread that Services Create/Delete RPC’s from the Engine</td>
</tr>
<tr>
<td>Skeleton</td>
<td>Remote Thread that Services Process/Control RPC’s for Codecs</td>
</tr>
<tr>
<td>Stub</td>
<td>Function that Marshalls RPC Arguments for Transport over DSP Link</td>
</tr>
<tr>
<td>VISA API</td>
<td>Functions to interface to xDM-compliant codecs using CE framework</td>
</tr>
<tr>
<td>xDAIS</td>
<td>eXpress DSP Algorithm Interface Standard. Used to instantiate algos</td>
</tr>
<tr>
<td>xDM</td>
<td>Interface that extends xDAIS, adding process and control functionality</td>
</tr>
</tbody>
</table>
(Optional) xDAIS Data Structures

The Param Structure

**Purpose**: To allow the application to specify to the algorithm the desired modes for any options the algorithm allows, e.g.: size of arrays, length of buffers, Q of filter, etc…

- `sizeof()`
- `filterType`
- `filterOrder`
- `bufferSize`
- ...
- ...

**Defined by**: Algorithm  
(in header file)

**Allocated by**: Application

**Written to by**: Application

**Read from by**: Algorithm

---

**Param Structures Defined in IMOD.H**

// IFIR_Params - structure defines instance creation parameters
typedef struct IFIR_Params {
    Int size;        /* 1st field of all params structures */
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
    XDAS_Int16 * coeffPtr;
} IFIR_Params;

// IFIR_Status - structure defines R/W params on instance
typedef struct IFIR_Status {
    Int size;        /* 1st field of all status structures */
    XDAS_Int16 blockSize;
    XDAS_Int16 * coeffPtr;
} IFIR_Status;
#include <std.h>
#include "ifir.h"

IFIR_Params IFIR_PARAMS = {
    sizeof(IFIR_Params),
    32,
    1024,
    0,
};

User may replace provided IFIR.C defaults with their preferred defaults
After defaults are set, Params can be modified for instance specific behavior

#include "ifir.h"
IFIR_Params IFIR_params ;
Create a Param structure
IFIR_params = IFIR_PARAMS;
Put defaults in Param structure
IFIR_params.firLen = 64;
Override length parameter
IFIR_params.blockSize = 1000;
Override block size parameter

The MemTab Structure

Purpose: Interface where the algorithm can define its memory needs and the application can specify the base addresses of each block of memory granted to the algorithm

Defined by: IALG Spec & Algorithm (rtm value of algNumAlloc)
Allocated by: Application
Written to by: Algorithm (4/5) & Application (base addr)
Read from by: Application (4/5) & Algorithm (base addr)

The Instance Object Structure

Purpose: To allow the application to specify to the algorithm the desired modes for any options the algorithm allows, eg: size of arrays, length of buffers, Q of filter, etc…

Defined by: Algorithm
Allocated by: Application via memRec(0) description
Written to by: Algorithm
Read from by: Algorithm (private structure!)
The vTab Concept and Usage

```
#include <ialg.h>
typedef struct IFIR_Handler {
    IALG_Fxns alg; /* IFIR extends IALG */
    Void (*filter)(IFIR_Handle handle, XDAS_In18 in[], XDAS_Out18 out[]);
} IFIR_Handler;

hFir->fxns->algInit((IALG_Handle)hFir, memTab, NULL,(IALG_Params *)&firParams);
```

```
#include <ialg.h>
typedef struct IFIR_Fxns {
    IALG_Fxns alg; /* IFIR extends IALG */
    Void (*filter)(IFIR_Handle handle, XDAS_Int8 in[], XDAS_Int8 out[]);
} IFIR_Fxns;
```

```
#include <ialg.h>
typedef struct IALG_Fxns {
    Void *implementationId;
    Void (*algActivate) (...);
    Int (*algAlloc) (...);
    Int (*algControl) (...);
    Void (*algDeactivate) (...);
    Int (*algFree) (...);
    Int (*algInit) (...);
    Void (*algMoved) (...);
    Int (*algNumAlloc) (...);
} IALG_Fxns;
```

vTab Structure

```
typedef struct IALG_Fxns {
    Void *implementationId;
    Void (*algActivate) (...);
    Int (*algAlloc) (...);
    Int (*algControl) (...);
    Void (*algDeactivate) (...);
    Int (*algFree) (...);
    Int (*algInit) (...);
    Void (*algMoved) (...);
    Int (*algNumAlloc) (...);
} IALG_Fxns;
```

Pragmas - For Linker Control of Code Sections

```
#pragma CODE_SECTION(FIR_TTO_activate, "text:algActivate")
#pragma CODE_SECTION(FIR_TTO_alloc, "text:algAlloc")
#pragma CODE_SECTION(FIR_TTO_control, "text:algControl")
#pragma CODE_SECTION(FIR_TTO_deactivate, "text:algDeactivate")
#pragma CODE_SECTION(FIR_TTO_free, "text:algFree")
#pragma CODE_SECTION(FIR_TTO_initObj, "text:algInit")
#pragma CODE_SECTION(FIR_TTO_moved, "text:algMoved")
#pragma CODE_SECTION(FIR_TTO_numAlloc, "text:algNumAlloc")
#pragma CODE_SECTION(FIR_TTO_filter, "text:filter")
```
Linker Control of Code Sections

- Users can define, with any degree of specificity, where particular algorithm components will be placed in memory.

| .text:algActivate  | IRAM |
| .text:algDeactivate | IRAM |
| .text:filter       | IRAM |
| .text              | SDRAM |

- Components not used may be discarded via the "NOLOAD" option.

| .text:algActivate  | IRAM |
| .text:algDeactivate | IRAM |
| .text:filter       | IRAM |
| .text:algAlloc     | SDRAM, type = NOLOAD |
| .text:algControl   | SDRAM, type = NOLOAD |
| .text:algFree      | SDRAM, type = NOLOAD |
| .text:algMoved     | SDRAM, type = NOLOAD |
| .text:algNumAlloc  | SDRAM, type = NOLOAD |
| .text              | SDRAM |

(Optional) Multi-Instance Ability
Multiple Instances of an Algorithm

Allocate, Activate as many instances as desired
Uniquely named handles allow control of individual instances of the same algorithm

All instance objects point to the same vtab
Coefficient array can be shared
Scratch can be separate or common as desired
(Optional) xDAIS : Static vs Dynamic

**xDAIS : Static vs. Dynamic**

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>SINE_init</td>
<td>algNumAlloc algAlloc algInit (aka sineInit)</td>
</tr>
<tr>
<td>Process</td>
<td>SINE_value SINE_blockFill</td>
<td>SINE_value SINE_blockFill</td>
</tr>
<tr>
<td>Delete</td>
<td>- none -</td>
<td>algFree</td>
</tr>
</tbody>
</table>

- Static usage requires programmer to read algo datasheet and assign memory manually.
- Codec Engine only uses the Dynamic features of xDAIS.

```plaintext
Dynamic (top) vs Static (bottom)

1. n = fxns->ialg.algNumAlloc();  //Determine number of buffers required
   memTab = (IALG_MemRec *)malloc (n*sizeof(IALG_MemRec));  //Build the memTab
   n = fxns->ialg.algAlloc((IALG_Params *)params,&fxnsPtr,memTab);  //Inquire buffer needs from alg
2. for (i = 0; i < n; i++) {  //Allocate memory for algo
    memTab[i].base = (Void *)memalign(memTab[i].alignment, memTab[i].size); }
3. alg = (IALG_Handle)memTab[0].base;  //Set up handle & *fxns pointer
   alg->fxns = &fxns->ialg;
4. bfxns->ialg.algInit(alg, memTab, NULL, (IALG_Params *)params);  // initialize instance object

1. |IALG_MemRec memTab[1];  // Create table of memory requirements
   int buffer0[5];  // Reserve memory for instance object
2. memTab[0].base = buffer0;  // with 1st element pointing to object itself
3. ISINE_Handle sineHandle;  // Create handle to InstObj
   sineHandle = memTab[0].base;
   sineHandle->fxns = &SINE_TTO_ISINE;  // Setup handle to InstObj
   sineHandle->fxns->ialg.algInit((IALG_Handle)sineHandle,memTab, NULL, (IALG_Params *)&sineParams);
```

---

Appendix
Introduction

In this chapter DMA use by algorithms/codecs will be considered.

Learning Objectives

At the conclusion of this chapter, you should be able to:

◆ Describe the range of operations possible with the DMA
◆ Demonstrate how to use ACPY3 API to perform DMA operations
◆ Describe how iDMA defines the needs of a given algorithm
◆ Describe how DMAN3 manages DMA hardware
◆ Show how to control the behavior of DMAN3 via CFG file entries
Chapter Topics

Using EDMA3 and ACPY3 .................................................................14-1

What is a DMA? (Hardware memcpy) ..................................................14-3

EDMA3 Overview ..............................................................................14-3

DMA : Basics ......................................................................................14-4
   Basics ..................................................................................................14-4
   DMA Examples ..................................................................................14-5
   Basic ACPY3 (memcpy on steroids) ...................................................14-8

DMA : Advanced Basics ..................................................................14-12
   Event, Transfer, Action (ETA) ...........................................................14-12
   Advanced ACPY3 ............................................................................14-13

Working with Fixed Resources .........................................................14-14
   Create, Process (transfer), Delete .....................................................14-14
   Writing iDMA3 Functions (Algo Author) ...........................................14-16
   Configuring DMAN3 (System Integrator ... i.e. Algo User) ...............14-17
What is a DMA? (Hardware memcpy)

How To Move Blocks of Memory?

```
<table>
<thead>
<tr>
<th>mem1</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
</tr>
</thead>
</table>
```

```
mem2
```

C6000

CPU
- memcpy()
- Takes DSP MIPs

Enhanced DMA (EDMA)
- Direct Memory Access
- Can directly access memory
- No CPU MIPs

EDMA3 Overview

Multiple DMA’s: EDMA3 and QDMA

```
VPSS
Master Periph
```

```
EDMA3
(System DMA)
```

```
DMA
(sync)
```

```
QDMA
(async)
```

```
C64x+ DSP
```

```
L1P
L1D
L2
```

Drivers

Algo’s

Both Share (number depends upon specific device)
- 128-256 Parameter RAM sets (PARAMs)
- 64 transfer complete flags
- 2-4 Pending transfer queues
DMA: Basics

DMA: Direct Memory Access

Goal:
- Copy from memory to memory

Examples:
- Import raw data from off-chip to on-chip before processing
- Export results from on-chip to off-chip afterward

Controlled by:
- Transfer Configuration (i.e. Parameter Set - aka PaRAM or PSET)
- Transfer configuration primarily includes 8 control registers

Transfer Configuration

How Much to Move?

“C” Count

Transfer Configuration

Options
Source
B
A
Destination
Index
Cnt Reload
Link Addr
Index
Resvd
C

B Count (# Elements)
A Count (Element Size)

C Count (# Frames)
DMA Examples

EDMA Example : Simple (Horizontal Line)

Goal:
Transfer 4 elements from loc_8 to myDest

Goal:
Transfer 4 elements from loc_8 to myDest

- DMA always increments across ACNT fields
- B and C counts must be 1 (or more) for any actions to occur

Here, ACNT was defined as element size : 1 byte
Therefore, BCNT will now be framesize : 4 bytes
B indexing must now be specified as well

Note: Less efficient version
EDMA Example: Indexing (Vertical Line)

Goal:
Transfer 4 vertical elements from loc_8 to a port

<table>
<thead>
<tr>
<th>loc_8 (bytes)</th>
<th>myDest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td></td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td></td>
</tr>
</tbody>
</table>

- ACNT is again defined as element size: 1 byte
- Therefore, BCNT is still framesize: 4 bytes
- SRCBIDX now will be 6 – skipping to next column
- DSTBIDX now will be 2

<table>
<thead>
<tr>
<th>Source</th>
<th>= &amp;loc_8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 =</td>
<td>BCNT</td>
</tr>
<tr>
<td>2 =</td>
<td>DSTBIDX</td>
</tr>
<tr>
<td>0 =</td>
<td>DSTCIDX</td>
</tr>
<tr>
<td></td>
<td>CCNT = 1</td>
</tr>
</tbody>
</table>

EDMA Example: Block Transfer (less efficient)

Goal:
Transfer a 5x4 subset from loc_8 to myDest

<table>
<thead>
<tr>
<th>16-bit Pixels</th>
<th>myDest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>2 8 9 10 11 12</td>
<td></td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td></td>
</tr>
</tbody>
</table>

- ACNT is defined here as 'short' element size: 2 bytes
- BCNT is again framesize: 4 elements
- CCNT now will be 5 – as there are 5 frames
- SRCCIDX skips to the next frame

<table>
<thead>
<tr>
<th>Source</th>
<th>= &amp;loc_8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 =</td>
<td>BCNT</td>
</tr>
<tr>
<td>2 =</td>
<td>DSTBIDX</td>
</tr>
<tr>
<td>2 =</td>
<td>DSTCIDX</td>
</tr>
<tr>
<td></td>
<td>CCNT = 5</td>
</tr>
</tbody>
</table>
**EDMA Example: Block Transfer (more efficient)**

**Goal:**
Transfer a 5x4 subset from loc_8 to myDest

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><strong>Elem 1</strong></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td><strong>Elem 2</strong></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td><strong>Elem 3</strong></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td><strong>Elem 4</strong></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td><strong>Elem 5</strong></td>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **myDest:**
  - 8
  - 9
  - 10
  - 11
  - 14
  - 15

16-bit Pixels

- ACNT is defined here as the entire frame: 4 * 2 bytes
- BCNT is the number of frames: 5
- CCNT now will be 1
- SRCBIDX skips to the next frame

Source
- 5 = &loc_8
- BCNT
- ACNT = 8
- DSTBIDX
- SRCBIDX

Destination
- 0 = &myDest
- DSTCIDX
- SRCCIDX = 0
- CCNT = 1

(4*2) is 8 = DSTCIDX SRCCIDX = 12 is (6*2) (from block 8 to 14)
Basic ACPY3 (memcpy on steroids)

### ACPY3 Method Chronology

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPY3_init()</td>
<td>Read to do ACPY3 work</td>
</tr>
<tr>
<td>ACPY3_configure()</td>
<td>Describe DMA actions to perform</td>
</tr>
<tr>
<td>ACPY3_start()</td>
<td>Begin DMA work</td>
</tr>
<tr>
<td>ACPY3_wait()</td>
<td>Optional CPU jobs in // w DMA</td>
</tr>
<tr>
<td>ACPY3_configure16()</td>
<td>Spin until DMA completes</td>
</tr>
<tr>
<td>ACPY3_start()</td>
<td>Other CPU activity - if desired</td>
</tr>
<tr>
<td>ACPY3_wait()</td>
<td>Optional CPU jobs in // w DMA</td>
</tr>
</tbody>
</table>

### ACPY3_configure

```c
extern void ACPY3_configure (IDMA3_Handle hdl
ACPY3_PaRam *PaRam, short transferNo);
```

**ACPY3 _configure must be called at least once for each individual transfer in a logical channel prior to starting the DMA transfer using ACPY3_start()...**

**ACPY3 PaRam:**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Options</th>
<th>TransferType</th>
<th>transferType</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>ACNT</td>
<td></td>
<td>1D1D, 1D2D, 2D1D or 2D2D</td>
<td></td>
</tr>
<tr>
<td>DSTBIDX</td>
<td>SRCBIDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCNTRLD</td>
<td>LINK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSTDIDX</td>
<td>SRCDDIDX</td>
<td></td>
<td>1D1D, 1D2D, 2D1D or 2D2D</td>
<td></td>
</tr>
<tr>
<td>- rsvd -</td>
<td>CCNT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ACPY3_PaRam:**

- **srcAddr**: Void *
- **dstAddr**: Void *
- **elementSize**: MdInt
- **numElements**: MdInt
- **numFrames**: MdInt
- **srcElementIndex**: MdInt
- **dstElementIndex**: MdInt
- **srcFrameIndex**: MdInt
- **dstFrameIndex**: MdInt
- **waitId**: MdInt
- **CCNT**: MdUns
- **BCNT**: MdUns
- **ACNT**: MdUns
- **SRCBIDX**: MdInt
- **SRCDDIDX**: MdInt
- **SRCCIDX**: MdInt
- **DSTDIDX**: MdInt
- **DSTBIDX**: MdInt
- **BCNTRLD**: MdInt
- **LINK**: MdInt
- **1D1D, 1D2D, 2D1D or 2D2D**: MdInt
- **-rsvd-**: MdInt

*discussed later*
**ACPY3_configure Example**

Goal:
Transfer 4 elements from loc_8 to myDest

<table>
<thead>
<tr>
<th>loc_8 (bytes)</th>
<th>myDest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7</td>
<td>8 9 10 11 12</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td>19 20 21 22 23 24</td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACPY3_TransferType</th>
<th>transferType</th>
<th>ACPY3_1D1D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void *</td>
<td>srcAddr</td>
<td>(IDMA3_AdrPtr) loc_8</td>
</tr>
<tr>
<td>Void *</td>
<td>dstAddr</td>
<td>(IDMA3_AdrPtr) myDest</td>
</tr>
<tr>
<td>MdUns</td>
<td>elementSize</td>
<td>4</td>
</tr>
<tr>
<td>MdUns</td>
<td>numElements</td>
<td>1</td>
</tr>
<tr>
<td>MdUns</td>
<td>numFrames</td>
<td>1</td>
</tr>
<tr>
<td>MdInt</td>
<td>srcElementIndex</td>
<td>0</td>
</tr>
<tr>
<td>MdInt</td>
<td>dstElementIndex</td>
<td>0</td>
</tr>
<tr>
<td>MdInt</td>
<td>srcFrameIndex</td>
<td>0</td>
</tr>
<tr>
<td>MdInt</td>
<td>dstFrameIndex</td>
<td>0</td>
</tr>
<tr>
<td>MdInt</td>
<td>waitId</td>
<td>-1 (discussed later)</td>
</tr>
</tbody>
</table>

**ACPY3_configure Example Code**

Goal:
Transfer 4 elements from loc_8 to myDest

<table>
<thead>
<tr>
<th>loc_8 (bytes)</th>
<th>myDest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7</td>
<td>8 9 10 11 12</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td>19 20 21 22 23 24</td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
</tbody>
</table>

```
ACPY3_PaRam PaRam;
PaRam.srcAddr = (IDMA3_AdrPtr)loc_8;
PaRam.dstAddr = (IDMA3_AdrPtr)myDest;
PaRam.transferType = IDMA3_1D1D;
PaRam.elemSize = 4;
PaRam.numElements = 1;
PaRam.numFrames = 1;
ACPY3_configure(hMyDma, &PaRam);
ACPY3_start(hMyDma);
```
### TransferType : 1D / 2D

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>ACPY3_PaRam Fields Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D1D</td>
<td>[--------]</td>
<td>[--------] elementSize</td>
</tr>
<tr>
<td>1D2D</td>
<td>[---][---]</td>
<td>[---][---] elementSize, numElements, dstElementIndex</td>
</tr>
<tr>
<td>2D1D</td>
<td>[---]</td>
<td>[---][---] elementSize, numElements, srcElementIndex</td>
</tr>
<tr>
<td>2D2D</td>
<td>[---]</td>
<td>[---][---] elementSize, numElements, srcElementIndex, dstElementIndex</td>
</tr>
</tbody>
</table>

*Where [--------] represents an "element" of `elementSize`*

*Obviously, all transfers require `srcAddr, dstAddr`*
ACPY3 Interface

<table>
<thead>
<tr>
<th>ACPY3 Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPY3_init</td>
<td>Initialize the ACPY3 module</td>
</tr>
<tr>
<td>ACPY3_activate</td>
<td>Activate individual DMA channel before using</td>
</tr>
<tr>
<td>ACPY3_configure</td>
<td>Configure a logical channel</td>
</tr>
<tr>
<td>ACPY3_fastConfigure16b</td>
<td>Modify a single (16-bit) PaRameter of the logical DMA channel</td>
</tr>
<tr>
<td>ACPY3_fastConfigure32b</td>
<td>Modify a single (32-bit) PaRameter of the logical DMA channel</td>
</tr>
<tr>
<td>ACPY3_start</td>
<td>Submit dma transfer request using current channel settings</td>
</tr>
<tr>
<td>ACPY3_wait</td>
<td>Wait for all transfers to complete on a specific logical channel</td>
</tr>
<tr>
<td>ACPY3_waitLinked</td>
<td>Wait for an individual transfer to complete on logical channel</td>
</tr>
<tr>
<td>ACPY3_complete</td>
<td>Check if the transfers on a specific logical channel have completed</td>
</tr>
<tr>
<td>ACPY3_completeLinked</td>
<td>Check if specified transfer on a specific logical channel have completed</td>
</tr>
<tr>
<td>ACPY3_setFinal</td>
<td>Specified transfer will be the last in a sequence of linked transfers</td>
</tr>
<tr>
<td>ACPY3_deactivate</td>
<td>Deactivate individual DMA channel when done using</td>
</tr>
<tr>
<td>ACPY3_exit</td>
<td>Free resources used by the ACPY3 module</td>
</tr>
</tbody>
</table>

ACPY3 fastConfigure32b, 16b

void ACPY3_fastConfigure32b (IDMA3_Handle handle, ACPY3_PaRamField32b fieldId, unsigned int value, short transferNo);

void ACPY3_fastConfigure16b (IDMA3_Handle handle, ACPY3_PaRamField16b fieldId, unsigned short value, short transferNo);

- This is a fast configuration function for modifying existing channel settings.
- Exactly one 16 (32) -bit channel transfer property, corresponding to the specified ACPY3_PaRam field, can be modified.
- Remaining settings of the channels configuration are unchanged.
Event, Transfer, Action (ETA)

- **Event** – triggers the transfer to begin
  - ACPY3 (async copy): User must trigger transfer, similar to memcpy()
  - For drivers, sync transfers are prog’d using the Low Level Driver (LLD) –discussed in a later chapter
- **Transfer** – the transfer config describes the transfers to be executed when triggered
- **Resulting Action** – what do you want to happen after the transfer is complete?
  - Two actions are available for ACPY3 transfers:
    - DMA sets a “done” (i.e. completed) flag which your program can test, or wait on
    - DMA triggers subsequent transfers … if a sequential list of transfers has been config’d

Sequential DMA Transfers

- **Transfer cfg:**
  - What to transfer?
  - Need to check if complete?
- **Linked sequential list:**
  - Use when transfer “1” must happen before transfer “2”
  - Even when transfers aren’t related, use to (slightly) increase init efficiency
  - “Complete” flags can occur after any/every transfer config (TrCfg) – as well as at the end of the list
- **Concurrent transfers:**
  - Multiple DMA transfers can be setup and triggered independently
  - Even though transfers may not occur simultaneously, it may help improve overall throughput

A DMA’s got to know it’s limitations …

- EDMA3 limits vary slightly for each processor
- Besides physical limits, ACPY3 shares some resources with synchronized (i.e. driver) copies
- Limits break into 3 categories:
  - Transfer config’s are stored in Parameter RAM (PaRAM) sets, which are limited
  - Limited # of transfer complete codes (i.e. “done” flags)
  - Concurrent sequences require multiple “channels”, which are limited
Advanced ACPY3

**ACPY3 Adv. Code Example**

```
#define tcfg0 0  //set transfer numbers
#define tcfg1 1

ACPY3_Params tcfg;

tcfg.transferType = ACPY3_1D1D;
tcfg.srcAddr = (IDMA3_AdrPtr) loc_start;
tcfg.dstAddr = (IDMA3_AdrPtr) loc_end;
tcfg.elementSize = 4 * sizeof (char);
tcfg.numElements = 1;
tcfg.numFrames = 1;
tcfg.waitId = 0;
ACPY3_configure (dmaHandle, &tcfg, tcfg0);

tcfg.srcAddr =  (IDMA3_AdrPtr) loc_end;
tcfg.dstAddr =  (IDMA3_AdrPtr) loc_start;
tcfg.waitId =  1;
ACPY3_configure (dmaHandle, &tcfg, tcfg1);
ACPY3_start (dmaHandle);
```

Transfer 4 bytes there, and back

```
Destination = &loc_end
Source = &loc_start
BCNT = 1  ACNT = 4
SRCID = 0  DSTID = 0
SRCBID = 0  DSTBID = 0
CCNT = 1
```

```
1 2 3 4 5 6
7 8 9 10 11
12 13 14 15 16
17 18 19 20 21
22 23 24 25 26
```

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

```c
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

```c
8 9 10 11 12
13 14 15 16 17
18 19 20 21 22
23 24 25 26 27
28 29 30 31 32
```

```c
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

```c
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

```c
8 9 10 11 12
13 14 15 16 17
18 19 20 21 22
23 24 25 26 27
28 29 30 31 32
```

```c
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

```c
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

```c
8 9 10 11 12
13 14 15 16 17
18 19 20 21 22
23 24 25 26 27
28 29 30 31 32
```
Working with Fixed Resources

**EDMA3 : Shared Resources**

- **EDMA3** (System DMA)
  - DMA (sync)
  - QDMA (async)

- **DMA**
  - Enhanced DMA (version 3)
  - DMA to/from peripherals
  - Can be sync'd to peripheral events
  - Handles up to 64 events

- **QDMA**
  - Quick DMA
  - DMA between memory
  - Async – must be started by CPU
  - Up to 8 channels available

Both Share
- 128-256 Parameter RAM sets (PARAMs)
- 64 transfer complete flags
- Limited pending transfer queues (various priorities)

Create, Process (transfer), Delete

**Solution to Limited Resources**

<table>
<thead>
<tr>
<th>Dynamic Memory Allocation</th>
<th>Dynamic Resource Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{malloc}(\text{size}); \text{<em>allocate</em>} )</td>
<td>Create Get DMA resource requirements</td>
</tr>
<tr>
<td>( x={\ldots}; \text{<em>initialize</em>} )</td>
<td>Initialize DMA</td>
</tr>
<tr>
<td>filter(...); \text{<em>execute</em>}</td>
<td>Process Use DMA Resources</td>
</tr>
<tr>
<td>free(a); \text{<em>release</em>}</td>
<td>Delete Release DMA resource</td>
</tr>
</tbody>
</table>

- Just as programmers have been using dynamic allocation to overcome limited memory size, we can do the same with fixed (i.e. limited) DMA resources
- TI's "framework components" provide two libraries for use with ACU3:
  - iDMA3 implemented by algorithm authors
  - DMAN3 called by application authors
DMAN3 / IDMA3 / ACPY3 Interaction

Application

Framework

DMA Manager Module (DMAN3)  ACPY3 Library

iDMA3 Interface

Algorithm

ACPY3 Provides support for configuring and instigating QDMA transfers (Code)
idMA3 Defines functions algo author must write to request QDMA resources (IF desc)
DMAN3 Application code which calls iDMA3 functions and allocates resources (Code)
**Working with Fixed Resources**

---

### iDMA3: Revisiting the Lifecycle of an Algo Instance

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Memory (iALG)</th>
<th>DMA (iDMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create</strong> (<em>&quot;Constructor&quot;</em>)</td>
<td>algNumAlloc algAlloc algInit</td>
<td>dmaGetChannelCnt dmaGetChannels dmaInit</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>algActivate doDSP (i.e., VIDDEC_process) algDeactivate</td>
<td>ACPY3_activate ACPY3_config ACPY3_start ACPY3_deactivate</td>
</tr>
<tr>
<td><strong>Delete</strong> (<em>&quot;Destructor&quot;</em>)</td>
<td>algFree</td>
<td>dmaGetChannels</td>
</tr>
</tbody>
</table>

- Great similarity between allocating Memory and DMA resources
- Like iALG, iDMA supports scratch sharing of resources between algo's

---

### Writing iDMA3 Functions (Algo Author)

#### Coding iDMA3 Functions (Algo Author)

```c
/* How many sets of DMA resources do I need? */
Uns FIR_TTO_dmaGetChannelCnt (Void) {
   return(1); // I want ONE
}

/* Fill in the DMA description table for each set of DMA resources you requested */
Uns FIR_TTO_dmaGetChannels (IALG_Handle handle, IDMA3_ChannelRec dmaTab[]) {
   FIR_TTO_Object *hObject = (Void *)handle;
   dmaTab[0].handle = hObject->dmaHandle;
   dmaTab[0].numTransfers = 2; // matches our previous ACPY3 example
   dmaTab[0].numWaits = 2;
   dmaTab[0].priority = IDMA3_PRIORITY_LOW;
   dmaTab[0].protocol = &ACPY3_PROTOCOL; // Using ACPY3 or your own code?
   dmaTab[0].persistent = FALSE;
   return (1);
}

/* Save handle to 'your' DMA resources ret'd by DMAN3 into the instance (i.e. class) object */
Int FIR_TTO_dmaInit (IALG_Handle handle, IDMA3_ChannelRec dmaTab[]) {
   FIR_TTO_Object *hObject = (Void *)handle;
   hObject->dmaHandle = dmaTab[0].handle;
   return (retval);
}
```
Configuring DMAN3 (System Integrator … i.e. Algo User)

DMAN3 Concepts

- DMAN3 module is very similar in function and usage to DSKT2
- Manages EDMA resources instead of Memory resources

Initialization Phase (build-time)

DMAN3

PaRam:
- #s 63-127

tcc:
- #s 32-63

QDMA chans

Usage Phase (run-time)

Alg1:
- 2 PaRam,
- 2 tcc
- 1 DMA ch

Alg2:
- 2 PaRam,
- 1 tcc
- 1 DMA ch

DMAN3 Setup: server.cfg

```javascript
var DMAN3 = xdc.useModule('ti.sdo.fc.dman3.DMAN3');

// set dman to use all external memory because video codecs take it all
//   note: scratch size of 60000 worked for mpeg and then leave internal
DMAN3.idma3Internal = false;
DMAN3.heapInternal = "L1DSRAM";
DMAN3.heapExternal = "DDR";
DMAN3.PaRamBaseIndex = 78;
DMAN3.numPaRamEntries = 48;

DMAN3.numQdmaChannels = 8;
DMAN3.qdmaChannels = [0, 1, 2, 3, 4, 5, 6, 7];
DMAN3.numPaRamGroup[0] = 48;
DMAN3.tccAllocationMaskL = 0;
DMAN3.tccAllocationMaskH = 0xff;
DMAN3.numTccGroup[0] = 8;

if (this.prog.build.profile == "debug")
    DMAN3.debug = true;
```

In addition to physical DMA resources, module needs some memory for storing PaRam shadows and other channel configuration states.

PaRam granted to the DMAN3 module by base index and number

Up to 8 QDMA channels available on 644x
tcc's are granted by bit mask
Appendix (More on EDMA3 Architecture)

EDMA3 Architecture

EVT's from peripherals/pins

Event Register (ER)
Evt Enable Reg (EER)
Event Set Reg (ESR)

Chain Event Reg (CER)
QDMA Evt Reg (QER)

64:1 Priority Encoder
8:1 Priority Encoder

QueMAP 0
QueMAP 1
QueMAP 71

MAP 0
MAP 1
.. MAP 71

PSET0
PSET1
.. PSET127

TC 0
TC 1
To SCR

Interrupt Pending Reg (IPR)
Interrupt Enable Reg (IER)

Completion

EDMAINT to CPU

EDMA : Adding Synchronization to DMA

Goal :
- Synchronized copy from port to memory

Examples :
- Accumulate new data from port to buffer before processing
- Export results from buffer to port a word at a time afterward

Synchronization Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Synchronization Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>reserved</td>
</tr>
<tr>
<td>2</td>
<td>ASP Rcv</td>
</tr>
<tr>
<td>3</td>
<td>ASP Xmt</td>
</tr>
<tr>
<td>4-7</td>
<td>VPSS</td>
</tr>
<tr>
<td>8-11</td>
<td>VICP</td>
</tr>
<tr>
<td>12-15</td>
<td>res'd</td>
</tr>
<tr>
<td>16-17</td>
<td>SPI</td>
</tr>
<tr>
<td>18-23</td>
<td>UART</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>52-54</td>
<td>PWM</td>
</tr>
<tr>
<td>55-64</td>
<td>res'd</td>
</tr>
</tbody>
</table>

Event Enable

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Enable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
</tr>
</tbody>
</table>

DMA job is placed on queue if EE=1

- Events can act as a trigger for A or B transfers (transfer all 3D with one event using DMA chaining)
- 1 event queues one transfer
- Allows multiple ports to be serviced concurrently by the EDMA engine
Introduction

This optional chapter provides a brief introduction to the DSP/BIOS Link protocol which provides a retargetable/portable interface between different processors. While this lower-level layer of target software is used by TI’s Codec Engine framework, it can also be used stand-alone.

Covered here are the basic architecture and the commonly used modules that make up the BIOS Link software, as well as a discussion of how the Codec Engine implements and extends the use of the Link protocol. Finally, a set of guidelines – along with a series of different use-cases – are provided to help you determine when it’s best to use the Codec Engine, or when it’s you may want to only implement the lower-level Link service.
Chapter Topics

(Optional) Introduction to DSP/BIOS Link

Introduction ................................................................................................................... 15-3
Where Have We Seen DSP/BIOS Link Before? .............................................................. 15-3
What is DSP/BIOS Link? .............................................................................................. 15-4
Codec Engine Advantages .......................................................................................... 15-5

Which Should I Use – Codec Engine vs DSP/BIOS Link .............................................. 15-6
Guidelines ...................................................................................................................... 15-6
Use Cases ...................................................................................................................... 15-7

DSP/BIOS Link Architecture ....................................................................................... 15-13

DSP/BIOS Link Modules .............................................................................................. 15-14
PROC ......................................................................................................................... 15-15
MSGQ .......................................................................................................................... 15-17
POOL .......................................................................................................................... 15-20
CHNL .......................................................................................................................... 15-22
(Optional) RINGIO ..................................................................................................... 15-24

What’s Next.................................................................................................................... 15-26

How DSPLink API’s are used by Codec Engine ............................................................ 15-27

For More Information ................................................................................................. 15-28
Introduction

Where Have We Seen DSP/BIOS Link Before?

Looking at the diagram that we’ve examined throughout most of this workshop, we can see that the DSP Link layer is a driver-level service that provides the intercommunication between both CPU’s.
What is DSP/BIOS Link?

Reviewing the description and features on this page, you may notice how similar Link is to the Codec Engine. Obviously, the Codec Engine does a good job using many of the various services provided by Link.

<table>
<thead>
<tr>
<th>What is DSP/BIOS™ LINK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Lower level inter processor communication link</td>
</tr>
<tr>
<td>◆ Allows master processor to control execution of slave processor</td>
</tr>
<tr>
<td>- Boot, Load, Start, Stop the DSP</td>
</tr>
<tr>
<td>◆ Provides peer-to-peer protocols for communication between the multiple processors in the system</td>
</tr>
<tr>
<td>- Includes complete protocols such as MSGQ, CHNL, RingIO</td>
</tr>
<tr>
<td>- Includes building blocks such as POOL, NOTIFY, MPCS, MPLIST, PROC_read/PROC_write which can be used to develop different kinds of frameworks above DSPLink</td>
</tr>
<tr>
<td>◆ Provides generic APIs to applications</td>
</tr>
<tr>
<td>- Platform and OS independent</td>
</tr>
<tr>
<td>◆ Provides a scalable architecture, allowing system integrator to choose the optimum configuration for footprint and performance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSPLink Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Hides platform/hardware specific details from applications</td>
</tr>
<tr>
<td>◆ Hides GPP operating system specific details from applications, otherwise needed for talking to the hardware (e.g. interrupt services)</td>
</tr>
<tr>
<td>◆ Applications written on DSPLink for one platform can directly work on other platforms/OS combinations requiring no or minor changes in application code</td>
</tr>
<tr>
<td>◆ Makes applications portable</td>
</tr>
<tr>
<td>◆ Allows flexibility to applications of choosing and using the most appropriate high/low level protocol</td>
</tr>
</tbody>
</table>
Codec Engine Advantages

Codec Engine (CE) provides many additional services:
- Memory, DMA/resource management
- Plug-n-play for XDM algorithms
- Supports DSP-only, GPP + Coprocessor, or GPP-only system
- Remote cache management
- Address Translation (ARM to DSP)
- Power Management (LPM)

Bottom line
- When using the DSP to execute algorithms called from the ARM CPU, the Codec Engine provides a greater ease of use due to it’s simple (but powerful) API, as well as the many additional services it provides.
- Accessing DSP/BIOS Link directly is more appropriate in those cases where additional flexibility is needed; for example, when you want to run two independent programs on the two CPU’s and just send some minor communications between them.
Which Should I Use – Codec Engine vs DSP/BIOS Link

Guidelines

Guidelines for Choosing IPC

◆ Codec Engine
  • When using XDAIS based Algorithms
  • Using multiple algorithms (or instances) on the DSP
  • Using the DSP as a the “ultimate” programmable H/W accelerator
  • If migration from one platform to another is needed
  • You prefer a structured, modular approach to software and want to leverage as much TI software as possible
  • When application runs algorithms locally

◆ DSPLink
  • When running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM) – often the case when using DSP-side I/O (i.e. DSP-based drivers)
  • When communicating between two discrete processors over PCI

Note
While it is possible to use both the Codec Engine and DSPLink simultaneously, this use-case is rarely required. Further it requires knowledge of Codec Engine, DSPLink, and LAD.
Use Cases

<table>
<thead>
<tr>
<th>Use Case #1</th>
</tr>
</thead>
</table>
| **Request:**
  
  I'm using DVSDK 2.0 on DM6446 - I want to add another audio algorithm.

| **Suggestion:**
  
  Codec Engine |

| **Guidelines:**
  
  - Using xDAIS based algorithm (xDM audio)
  - Also, using DSP as algorithm accelerator to the ARM/Linux program |

| **Notes:**
  
  - DVSDK includes video, speech, audio, image and universal interfaces
  - Details on how to integrate new codecs into DVSDK:
    
Use Case #2

Request:
I'm using OMAP35x - I want to add a bar-code scanner algo on the DSP

Suggestion:
Codec Engine – if algorithm is xDAIS compliant or using multiple ones
DSPLink – if using single, non-compliant algorithm

Guidelines:
- Multiple – see provided flowchart

Notes:
- Using Codec Engine eases burden for ARM/Linux programmer, but requires algorithm author to package DSP algo into a xDAIS/xDM class
- DSPLink provides lower-level interface (simpler architecture), but does not manage resources which makes sharing between algorithms more difficult.

Use Case #2 – Guideline Flowchart
Use Case #3

Request:
I'm using an OMAP-L138 - I want to build my own DSP-side application and use DSP side I/O

Suggestion:
DSPLink

Guidelines:
• Running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM)

Notes:
• ARM is not using DSP as an algorithm accelerator
• Example:
  • Industrial application where ultra-low latency I/O drivers and processing is critical
  • Only req's a few "control cmds" from the ARM-side to influence the DSP processing
• Since this use-case does not require additional services of Codec Engine, the less-complicated DSPLink architecture may be preferred
• If multiple DSP algo's are needed, but being called from the DSP, you might choose to use a "local" Codec Engine
• An example application showing this is part of OMAP-L138 SDK release

Use Case #4

Request:
I'm on DM6446 but I plan to migrate to DM365 later - I want to keep the same APIs.

Suggestion:
Codec Engine !

Guidelines:
• Using xDAIS based algorithm (xDM video/imaging/speech/audio)
• If migrating from one platform to another

Notes:
• Codec Engine is exactly the right fit here as CE has the same APIs regardless of whether you are on an ARM-only, DSP-only, or ARM+DSP
• CE provides needed resource management services to share memory, DMA, co-processors effectively
Use Case #5

Request:

I have an audio/video system - I want to add another non-VISA codec.

Suggestion:

Codec Engine – add algorithm using VISA’s iUNIVERSAL API

Guidelines:

- Using multiple algorithms (or instances) on the DSP

Notes:

- Codec Engine framework can be extended to support add’l algorithms
- Example: Bit-blit algorithm has been supported using IUNIVERSAL APIs
  - [Bitblit](https://gforge.ti.com/gf/project/bitblit/)

Use Case #6

Request:

I am using ARM/x86 (GPP) and multiple DM6437 (DSP) devices in my system connected over PCI. How can I pass information between these discrete processors?

Suggestion:

DSPLink

Guidelines:

- When communicating between discrete processors over PCI
- Running a stand-alone DSP program

Notes:

- Currently, CE only supports ARM+DSP SOC devices (using shared-memory) since shared memory allows for fast data-sharing (i.e. pointer passing) between CPU’s.
- DSP and ARM each manage their own I/O; usually IPC is only needed to pass control/command information, as opposed to streaming data
- It's likely that you will use CE on each processor to easily implement 'local' algorithms; but DSPLink would be used for IPC
Use Case #7

Request:
I am using DM355 processor and plan to run audio algorithm on ARM.

Suggestion:
Codec Engine

Guidelines:
- When application run algorithms locally

Notes:
- Codec Engine supports running algorithms locally on GPP only (DM355, DM365) or DSP only devices (DM6437, DM648)
- DSP Link is used for communication between discrete processors and not useful for GPP only or DSP only implementation.

Use Case #8

Request:
I am using OMAP-L138 processor for calling audio algorithm and in addition, I want to get messages from DSP task that is doing I/O to ARM.

Suggestion:
Codec Engine + DSPLink + Link Arbiter Daemon (LAD)

Guidelines:
- When using XDAIS based Algorithms
- When running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM) – often the case when using DSP-side I/O (i.e. DSP-based drivers)

Notes:
- Codec Engine is used to call XDAIS audio algorithm.
- DSP Link is used to exchange messages with DSP task that is doing I/O separately from the DSP Server and codecs combinations.
- Link Arbiter Daemon (LAD) is needed to support running Codec Engine and DSPLink simultaneously

http://tiexpressdsp.com/index.php/Link_Arbiter_Daemon
## IPC Use Case Summary

<table>
<thead>
<tr>
<th>#</th>
<th>Use Case</th>
<th>CE / Link</th>
<th>Device / Family</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add xDM algo</td>
<td>dvsdk</td>
<td>DM6446</td>
<td>use ce</td>
</tr>
<tr>
<td>2</td>
<td>Add Bar code algo</td>
<td>Algorithm not part of DVSDK</td>
<td>OMAP35x</td>
<td>Use IUNIVERSAL APIs</td>
</tr>
<tr>
<td>3</td>
<td>Separate ARM/Linux and DSP programs</td>
<td>DSPLink</td>
<td>OMAP-L1x</td>
<td>For control commands &amp; messages</td>
</tr>
<tr>
<td>4</td>
<td>Want to migrate (i.e. port) later on</td>
<td>Codec Engine</td>
<td>dm355 going to omap35x</td>
<td>CE provides migration from GPP only to GPP+DSP</td>
</tr>
<tr>
<td>5</td>
<td>A/V system and want to add non-VISA algo</td>
<td>Codec Engine</td>
<td>DM6446, OMAP35x</td>
<td>Use IUNIVERSAL APIs</td>
</tr>
<tr>
<td>6</td>
<td>DSP-side I/O</td>
<td>DSPLink</td>
<td>OMAP-L1x</td>
<td>Simpler architecture</td>
</tr>
<tr>
<td>7</td>
<td>Multiple algorithms</td>
<td>Codec Engine</td>
<td>DM6446/DM3xx/OMAP3</td>
<td>Multiple XDAIS algorithms</td>
</tr>
<tr>
<td>8</td>
<td>xDM algos and own bios program doing DSP I/O</td>
<td>CE + DSPLink + LAD</td>
<td>OMAP-L1x</td>
<td>Preferred APIs</td>
</tr>
</tbody>
</table>
DSP/BIOS Link Architecture

Architecture

[Diagram showing the architecture of DSP/BIOS Link, including GPP, OS, LINK DRIVER, APPLICATION / FRAMEWORK, and other components.]

- DSP/BIOS Link generic modules
- DSP/BIOS LINK generic modules
- DSP/BIOS LINK port-specific modules
- Drivers
- Application
- GPP OS
### DSP/BIOS Link Modules

#### Modules - Overview

- **Device Manager for DSP control**
  - **PROC**: Boot-load, start, stop the DSP

- **Inter-Processor Communication protocols**
  - Complete protocols for different types of data transfer between the processors
  - Each protocol meets a different data transfer need
  - **MSGQ**: Message Queue
  - **CHNL**: SIO/streaming based on issue-reclaim model
  - **RingIO**: Circular ring buffer

- **Inter-Processor Communication building blocks**
  - Low-level building blocks used by the protocols
  - Each building block is also exposed as APIs to allow framework writers to define their own application-specific protocols
  - **POOL**: Memory Manager - shared/non-shared
  - **NOTIFY**: Interrupt abstraction and de-multiplexing
  - **MPCS**: Multi-processor critical section
  - **MPLIST**: Multi-processor doubly linked circular list
  - **PROC read & PROC write**: Read from or write to DSP memory
PROC

PROC: DSP Boot-loading

- DSP executable is present in the GPP file system
- The specified executable is loaded into DSP memory (internal/external)
- The DSP execution is started at its entry point
- Boot-loading using: Shared memory, PCI, etc.

PROC: Write/Read

- PROC_write
  Write contents of buffer into specified DSP address
- PROC_read
  Read from specified DSP address into given buffer
- Can be used for very simple data transfer in static systems
- Application Example on using PROC_read and PROC_write
## PROC APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC_setup</td>
<td>Setup PROC sub-component</td>
</tr>
<tr>
<td>PROC_attach</td>
<td>Attach to specific DSP and initialize it</td>
</tr>
<tr>
<td>PROC_load</td>
<td>Load the specific base image on target DSP</td>
</tr>
<tr>
<td>PROC_start</td>
<td>Starts the execution of loaded code on DSP</td>
</tr>
<tr>
<td>PROC_control</td>
<td>Provides a hook to perform device dependent control operations</td>
</tr>
<tr>
<td>PROC_debug</td>
<td>Prints the current status of this component for debugging purposes</td>
</tr>
<tr>
<td>PROC_loadSection</td>
<td>Load particular section from base image</td>
</tr>
<tr>
<td>PROC_destroy</td>
<td>Destroys PROC sub-component</td>
</tr>
<tr>
<td>PROC_detach</td>
<td>Detach the DSP</td>
</tr>
<tr>
<td>PROC_stop</td>
<td>Stops the execution on the target DSP</td>
</tr>
</tbody>
</table>

## PROC APIs (Contd..)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC_getState</td>
<td>Get current state of the target DSP</td>
</tr>
<tr>
<td>PROC_read</td>
<td>Reads from specific addresses of the target DSP</td>
</tr>
<tr>
<td>PROC_write</td>
<td>Writes to specific addresses of the target DSP</td>
</tr>
<tr>
<td>PROC_instrument</td>
<td>Gets instrumentation data associated with PMGR_PROC sub-component</td>
</tr>
<tr>
<td>PROC_isAttached</td>
<td>Checks if the target DSP is attached</td>
</tr>
</tbody>
</table>
MSGQ

MSGQ: Overview

- Messaging protocol allows clients to send messages to a named Message Queue located on any processor on the system
- Message Queue can have: single reader, multiple writers

MSGQ: Features

- Messaging provides logical connectivity between GPP and DSP clients
- Messages are sent at a higher priority than CHNL data buffers by default (Configurable)
- Messages can be variable sized
- Messages can be sent to a named Message Queue
- Message Queues have unique system-wide names. Senders locate the Message Queue using this name to send messages to it.
- Client can send messages to Message Queues that are created on any processor connected to the local processor using a transport
**MSGQ: Data Flow Example**

**GPP**
- MSGQ_locate("DSPMSGQ1", ...)
- MSGQ_alloc()
- MSGQ_put (DSPMSGQ1, ...)
- MSGQ_release (DSPMSGQ1, ...)

**DSP**
- MSGQ_open("DSPMSGQ1", ...)
- MSGQ_get (DSPMSGQ1, ...)
- MSGQ_free (...)
- MSGQ_close (DSPMSGQ1, ...)

This sequence is also valid for Message Queue created on GPP-side

Message Queue *DSPMSGQ1*

Writer: DSPMSGQ1

Reader: DSPMSGQ1
### MSGQ APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGQ_open</td>
<td>Opens message queue to be used for receiving messages</td>
</tr>
<tr>
<td>MSGQ_close</td>
<td>Closes the message queue</td>
</tr>
<tr>
<td>MSGQ_locate</td>
<td>Synchronously locates the message queue identified by specified MSGQ name</td>
</tr>
<tr>
<td>MSGQ_release</td>
<td>Releases the message queue that was located earlier</td>
</tr>
<tr>
<td>MSGQ_alloc</td>
<td>Allocates a message</td>
</tr>
<tr>
<td>MSGQ_free</td>
<td>Frees a message</td>
</tr>
<tr>
<td>MSGQ_put</td>
<td>Sends a message to specified Message queue</td>
</tr>
<tr>
<td>MSGQ_get</td>
<td>Receives a message on specified message queue</td>
</tr>
<tr>
<td>MSGQ_setErrorHandler</td>
<td>Allows the user to designate MSGQ as an error handler MSGQ to receive asynchronous error messages from the transports</td>
</tr>
<tr>
<td>MSGQ_count</td>
<td>Returns count of number of messages in a local message queue</td>
</tr>
</tbody>
</table>

### MSGQ APIs (Contd..)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGQ_transportOpen</td>
<td>Initializes transport associated with the specified processor</td>
</tr>
<tr>
<td>MSGQ_getSrcQueue</td>
<td>Gets source message queue of a message to be used for replying to the message</td>
</tr>
<tr>
<td>MSGQ_debug</td>
<td>Prints current status of MSGQ sub-component</td>
</tr>
<tr>
<td>MSGQ_locateAsync</td>
<td>Asynchronously locates the message queue</td>
</tr>
<tr>
<td>MSGQ_transportClose</td>
<td>Closes the transport associated with the specified processor</td>
</tr>
<tr>
<td>MSGQ_instrument</td>
<td>Gets instrumentation information related to specified message queue</td>
</tr>
</tbody>
</table>
POOL

POOL: Features

- Allows configuration of shared memory regions as buffer pools
- These buffers are used by other modules from DSPLink for providing inter-processor communication functionality.
- The specific services provided by this module are:
  - Configure shared memory region through open & close calls.
  - Allocate and free buffers from the shared memory region.
  - Translate address of a buffer allocated to different address spaces (e.g. GPP to DSP)
  - Synchronize contents of memory as seen by the different CPU cores.
- Provides a uniform view of different memory pool implementations, which may be specific to the hardware architecture or OS on which DSPLink is ported.
- This component is based on the POOL interface in DSP/BIOS™.
- APIs for SMA POOL in DSPLink are callable from TSK/SWI context on DSP-side. They must not be called from ISR context.

POOL: Overview

- Configures and manages shared memory regions across processors
- Supports multiple clients on GPP and DSP

<table>
<thead>
<tr>
<th>GPP</th>
<th>Shared Memory</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_open()</td>
<td></td>
<td>POOL_open()</td>
</tr>
<tr>
<td>POOL_alloc(256, ...)</td>
<td></td>
<td>POOL_alloc(1024, ...)</td>
</tr>
<tr>
<td>POOL_free(buf,...)</td>
<td></td>
<td>POOL_free(buf,...)</td>
</tr>
<tr>
<td>POOL_close()</td>
<td></td>
<td>POOL_close()</td>
</tr>
</tbody>
</table>
### POOL APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>POOL_open</code></td>
<td>Opens a specified pool referenced by the pool ID</td>
</tr>
<tr>
<td><code>POOL_close</code></td>
<td>Closes a specific pool</td>
</tr>
<tr>
<td><code>POOL_alloc</code></td>
<td>Allocates a buffer of specified size from a pool</td>
</tr>
<tr>
<td><code>POOL_free</code></td>
<td>Frees a buffer into the specified pool</td>
</tr>
<tr>
<td><code>POOL_translateAddr</code></td>
<td>Translates addresses between two address spaces for a buffer that was allocated from the pool</td>
</tr>
<tr>
<td><code>POOL_writeback</code></td>
<td>Writes the content of GPP buffer into DSP buffer (with offset in sync)</td>
</tr>
<tr>
<td><code>POOL_invalidate</code></td>
<td>Invalidates the contents of the buffer</td>
</tr>
</tbody>
</table>
CHNL

CHNL: Overview

- Issue/reclaim protocol as used by DSP/BIOS™ SIO module.
- Single reader, single writer

```
CHNL_issue ()  CHNL_reclaim ()
CHNL_issue ()  CHNL_reclaim ()
```

<table>
<thead>
<tr>
<th>Data Source (I/P Channel)</th>
<th>Data Destination (O/P Channel)</th>
</tr>
</thead>
</table>

Buffer exchange

Filled Buffers
Empty Buffers

This sequence is also valid for output channel on GPP and input channel on DSP

CHNL: Features

- Data Channel is a virtual entity providing interface to send/receive data buffers over a physical connection
- Multiple data channels are multiplexed over the same physical link
- Channels are uni-directional
- Multiple buffers can be queued on the channel for better performance
### CHNL APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHNL_create</td>
<td>Create resources used for transferring data between GPP and DSP</td>
</tr>
<tr>
<td>CHNL_delete</td>
<td>Release channel resources used for transferring data between GPP and DSP</td>
</tr>
<tr>
<td>CHNL_allocateBuffer</td>
<td>Allocates an array of buffers of specified size</td>
</tr>
<tr>
<td>CHNL_freeBuffer</td>
<td>Frees buffer that were previously allocated</td>
</tr>
<tr>
<td>CHNL_issue</td>
<td>Issues an input or output request on a specified channel</td>
</tr>
<tr>
<td>CHNL_reclaim</td>
<td>Gets the buffer back that has been issued to this channel</td>
</tr>
<tr>
<td>CHNL_idle</td>
<td>Resets the input stream channel. Waits/idles the output stream channel for as long as it takes to transfer currently queued buffers</td>
</tr>
<tr>
<td>CHNL_flush</td>
<td>Discards all requested buffers pending for transfer in both input and output channels</td>
</tr>
<tr>
<td>CHNL_control</td>
<td>Provides a hook to perform device dependent control operations</td>
</tr>
<tr>
<td>CHNL_debug</td>
<td>Prints current status of PMGR_CHNL sub-component</td>
</tr>
</tbody>
</table>
(Optional) RINGIO

RINGIO: Overview

- Circular Ring Buffer based data streaming, optimized for audio/video processing
- Single reader, single writer

RINGIO: Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides true circular buffer view to the application</td>
<td>Internally handles wraparound</td>
</tr>
<tr>
<td>Data and attributes/messages associated with data can be sent</td>
<td>In-band attributes are supported</td>
</tr>
<tr>
<td>Reader and writer can work on different data sizes</td>
<td>Use case: Writer can be unaware of reader buffer size needs</td>
</tr>
<tr>
<td>Reader and writer can operate completely independent of each other</td>
<td>Synchronization only needed when acquire fails due to insufficient available size</td>
</tr>
<tr>
<td>Capability to minimize interrupts by choosing specific notification type</td>
<td>Notification configurable based on threshold and type (ONCE/ALWAYS)</td>
</tr>
<tr>
<td>Ability to cancel unused (acquired but unreleased) data</td>
<td>Use case: Simultaneously work on two frames: previous and new</td>
</tr>
<tr>
<td>Ability to flush out released data</td>
<td>Use case: stop/fast forward</td>
</tr>
<tr>
<td>On DSP-side, APIs can be called from TSK or SWI context. (Though, they must not be called from ISR context.)</td>
<td>Flexibility in DSP-side choice of thread type</td>
</tr>
</tbody>
</table>
### RINGIO APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RingIO_getAcquiredOffset</td>
<td>Returns the current acquire offset for the client</td>
</tr>
<tr>
<td>RingIO_getAcquiredSize</td>
<td>Returns size of buffer current acquired by the client</td>
</tr>
<tr>
<td>RingIO_getWatermark</td>
<td>Returns the current watermark level specified by client</td>
</tr>
<tr>
<td>RingIO_create</td>
<td>Creates a RingIO instance in shared memory</td>
</tr>
<tr>
<td>RingIO_delete</td>
<td>Deletes a RingIO instance in shared memory</td>
</tr>
<tr>
<td>RingIO_open</td>
<td>Opens a RingIO instance handle</td>
</tr>
<tr>
<td>RingIO_close</td>
<td>Closes an already open RingIO reader/writer</td>
</tr>
<tr>
<td>RingIO_acquire</td>
<td>Acquires a data buffer from RingIO for reading or writing</td>
</tr>
<tr>
<td>RingIO_release</td>
<td>Releases a previously acquired buffer</td>
</tr>
<tr>
<td>RingIO_cancel</td>
<td>Cancels any data buffer acquired by reader or writer</td>
</tr>
</tbody>
</table>

### RINGIO APIs contd..

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RingIO_getAttribute</td>
<td>Gets a fixed size attribute from the attribute buffer</td>
</tr>
<tr>
<td>RingIO_setAttribute</td>
<td>Sets a fixed size attribute at the offset provided in acquired data buffer</td>
</tr>
<tr>
<td>RingIO_getvAttribute</td>
<td>Gets an attribute with a variable sized payload from the attribute buffer</td>
</tr>
<tr>
<td>RingIO_setvAttribute</td>
<td>Sets an attribute with a variable sized payload at the offset provided in acquired data buffer</td>
</tr>
<tr>
<td>RingIO_flush</td>
<td>Flush the data from the RingIO</td>
</tr>
<tr>
<td>RingIO_setNotifier</td>
<td>Sets Notification parameters for the RingIO client</td>
</tr>
<tr>
<td>RingIO_sendNotify</td>
<td>Sends a notification to the other client with an associated message value</td>
</tr>
<tr>
<td>RingIO_getValidSize</td>
<td>Returns valid size in the RingIO</td>
</tr>
<tr>
<td>RingIO_getValidAttrSize</td>
<td>Returns the valid attributes size in the RingIO</td>
</tr>
<tr>
<td>RingIO_getEmptySize</td>
<td>Returns the current empty data buffer size</td>
</tr>
<tr>
<td>RingIO_getEmptyAttrSize</td>
<td>Returns the current empty attribute buffer size</td>
</tr>
</tbody>
</table>
What’s Next

- Support for **Grid topology** (any to any connection)
  - Currently supports only star topology (one master, many slaves)
- **Portable architecture** to support completely different types of physical links like EMAC/USB
  - Currently, DSPLink is difficult to port to non-shared memory architectures except in PCI & VLYNQ.
- Available on main-line Linux kernel tree for Davinci/OMAP
- **Dynamic linking/loading; Dynamic memory-mapping**
- Add **Power management** support
How DSPLink API’s are used by Codec Engine

**Engine – CODEC Engine – DSP Link**

VISA API Layer: Application Programmer

- Engine_open()
- Engine_close()

CODEC Engine/DSP Link framework: TI

- PROC_setup
- PROC_load
- PROC_attach
- PROC_start
- POOL_open
- MSGQ_TransportOpen
- MSGQ_alloc
- MSGQ_create
- MSGQ_locate

xDM API Layer: Algorithm Author

VISA – CODEC Engine – DSP Link

VISA API Layer: Application Programmer

- VIDDEC_create()
- VIDDEC_control()
- VIDDEC_process()
- VIDDEC_delete()

CODEC Engine/DSP Link framework: TI

- MSGQ_alloc
- MSGQ_put
- MSGQ_get
- MSGQ_free

- MSGQ_alloc
- MSGQ_put
- MSGQ_get
- MSGQ_free

- MSGQ_alloc
- MSGQ_put
- MSGQ_get
- MSGQ_free

- MSGQ_alloc
- MSGQ_put
- MSGQ_get
- MSGQ_free

xDM API Layer: Algorithm Author
For More Information

- **DSP Link User's Guide**
  (Part of DSPLink installation under docs folder)

- **DSP Link Programmer's Guide**
  (Part of DSPLink installation under docs folder)

- **DSPLink Wiki**
Appendix A: Questions & Answers

Table of Contents

Note, since not all lab exercises ask you written questions, there will be gaps between labs in this appendix.

Appendix A: Questions & Answers ............................................................................................................... 1-1

Lab 5 ....................................................................................................................................................... 1-2
Lab 6 ....................................................................................................................................................... 1-3
Lab 7 ....................................................................................................................................................... 1-4
Lab 8 ....................................................................................................................................................... 1-5
Lab 9 ....................................................................................................................................................... 1-6
Lab 10 ..................................................................................................................................................... 1-7
Lab 11 ..................................................................................................................................................... 1-7
Lab 12 ..................................................................................................................................................... 1-8
Lab Questions 5.1

1. Which file in the lab05a_hello_world/app directory specifies that one or more executable files are to be built?
   - package.bld

2. Which function call within this file actually scripts the building of a new executable file?
   - Pkg.addExecutable(...)

3. Examination of the config.bld file (at the top level of the workshop directory) shows that we have configured a third target, named C64P. This target is configured to build for the C64x+ DSP core that is available on the TMS320C6446. How could you modify the available files to build a Hello World executable that would run on the C64x+ core?
   - In package.bld change:
     ```
     var targs = [MVArm9, Linux86]
     ```
   - to:
     ```
     var targs = [MVArm9, Linux86, C64P]
     ```

4. (Advanced) Explain in your own words how the C source files used to build this application are specified in the package.bld build script of lab05b_extended_features.
   - The java.io.File('.').list() I/O method is used to get a listing of all files in the current directory. Then the list of files is iterated through and the String().match() method is used to filter out those files which end in ".c" using the regular expression /\^\.c$/

Lab Questions 5.2

5. When compiling code for the C64x+ DSP, what compiler options are used by in this lab exercise.

   **Hint: look at options specified in these files:**

   ```
   config.bld:  --no_compress --mem_model=data_far --disable:sploop "+ remarks"
   ```
   (xdc parameter)

   ```
   ti.targets.C64P:  -eo.o64P -ea.s64P -mv64p
   ```
   (specified in config.bld)

   ```
   ti.platforms.evmDM6446:  none
   ```
   (specified in config.bld)

   ```
   package.bld:  none
   ```
   (implicit xdc file)
Lab 6

Lab Questions 6.1

1. Which ioctl command is called in the init_mixer() function of audio_input_output.c in order to specify whether the microphone or line input will provide the audio source for the driver?

\[
\text{ioctl(mixerFd, SOUND_MIXER_WRITE_RECSRC, &recmask)}
\]

2. Which ioctl commands set the number of channels and sample rate for the OSS driver (two different ioctl commands are needed), and in which function of audio_input_output.c are they found?

\[
\begin{align*}
\text{ioctl(fd, SNDCTL_DSP_CHANNELS, &numchannels)} \\
\text{ioctl(fd, SNDCTL_DSP_SPEED, &samplerate)}
\end{align*}
\]

and they’re found in init_sound_device()

3. For the while() loop in the audio_thread_fxn of either the audio recorder or audio playback application: There is an (f)read and an (f)write function in this loop. For the file read or write function, a FILE pointer is the first parameter passed. For the driver read or write function, a file descriptor (int) is the first parameter passed. What is the purpose of the FILE pointer and file descriptor, and where do these come from? (In other words, what function is used to generate valid file descriptors and FILE pointers from which read and write operations can be made?)

The purpose of a FILE pointer or a file descriptor is a reference to a given file or device which is used to read from or write to the given resource. It is set during the (f)open call.

Lab Questions 6.2

4. (Advanced) The audio_input_setup and audio_output_setup functions take as their first parameter not an integer file descriptor, but rather a pointer to an integer. The name of the variable in these function prototypes is fdByRef, indicating that it is a variable passed by reference. Why is a pointer to integer used here when file descriptors are integers?

Because a pointer to the file descriptor is passed, i.e. the file descriptor is passed by reference, it's value can be modified by the function. Variables that are not passed by reference are passed by copy and any changes made by a function are made to the copy and not the original.

5. (Advanced) When the audio_input_setup function is called from the audio_thread_fxn of lab6a_audio_recorder, the inputFd variable is passed preceded by an ampersand. Why? (See question 4, which is related.)

Because the file descriptor is passed by reference, it's address must be passed instead of its value. The ampersand operator indicates the variable’s address.

6. What changes were made to the package.bld XDC build script between lab04_hello_world and lab06a_audio_recorder in order to generate two executables: one that is a debug version and one that is a release (optimized) version?

\[
\text{var profiles = ['release'];} \rightarrow \text{var profiles = ['release', 'debug'];}
\]
Lab 7

Lab 7.1

1. How would you modify the `lab07a_osd_setup` application to make the banner you
   created semi-transparent instead of solid?
   
   In the `video_osd_place` function, change the value of `trans` to `0x44`

2. How would you modify the `lab07a_osd_setup` application to place your banner at the
   top of the screen instead of the bottom?
   
   In the `video_osd_place` function, change the value of `y_offset` to `0`

3. Why is it necessary to run the `bmpToRgb16.x86U` utility on the bitmapped image that
   you created with GIMP before it could be displayed in the DM6446 on-screen-display
   window?
   
   The GIMP program uses 24-bit color (8 red, 8 green, 8 blue) per pixel bitmaps. The
   OSD window is 16-bit color (5 red, 6 green, 5 blue) per pixel.

4. Which ioctl command is called in `setup_video_input` function of `video_input.c` in order
   to specify NTSC (as opposed to PAL) as the video input standard?
   
   `ioctl(captureFd, VIDIOC_S_STD, &std)`

5. Which ioctl command is used to set the pixel width and height for the V4L2 input
   driver in `setup_video_input`?
   
   `ioctl(captureFd, VIDIOC_S_FMT, &fmt)`

Lab 7.2

6. `video_input_setup` uses the `VIDIOC_REQBUFS` ioctl to request the number of video
   input buffers desired to be allocated by the driver. The driver then uses
   `VIDIOC_QUERYBUF` in order to determine the length and offset of each buffer.
   Which function call then uses this length and offset? What is the purpose of this
   function call?
   
   The `mmap` function uses this information to map the buffers into the application’s
   memory space so that the application can access them directly.

7. What is the underlying (Linux) function utilized by `wait_for_frame` in `video_input.c` to
   block the application’s execution until the next frame of video data is available on the
   V4L2 driver?
   
   `select()`

8. Which ioctl command is used by the `flip_display_buffers` function to exchange a
   working video output buffer for the displayed video output buffer?
   
   `ioctl(displayFd, FBIO_PAN_DISPLAY, &vinfo)`

9. In addition to closing the video output device, what other cleanup is handled by the
   `video_output_cleanup` function?
   
   `munmap` function is used to unmap the driver buffers from the application’s memory
   space.
Lab 8

1. In lab08b_audio_video_rtime, main.c, what priority is the audio thread set to? (You may use an expression here.)
   \[
   \text{sched\_get\_priority\_max(SCHED\_RR)}
   \]

2. When running the debug version of the application (./app_debug.x470MV), what does the debug output indicate the audio thread priority is set to? (Numerical value)
   99

3. What priority is the video thread set to? (You may use an expression here.)
   \[
   \text{sched\_get\_priority\_max(SCHED\_OTHER)}
   \]

4. When running the debug version of the application (./app_debug.x470MV), what does the debug output indicate the video thread priority is set to? (Numerical value)
   0
Lab 9

Lab 9.1

1. Which VISA api function call is made in `video_encoder_setup`?
   - `VIDENC_create()`

2. Why is the encoderHandleByRef parameter that is passed to the `video_encoder_setup` function a pointer to a VIDENC_Handle instead of just a VIDENC_Handle?
   - So that the handle's value can be modified in order to return the handle to the newly created video encoder.

3. In the `inBufDesc` buffer descriptor used in the `encode_video` function, explain why the `.bufs` parameter is a pointer to a pointer to XDAS_Int8 (i.e. a pointer to a pointer to char).
   - The pointer to char value represents the address of a buffer. This is a pointer to pointer to char because it is an array of pointer to char, i.e. an array of buffer addresses.

4. Both the `audio_thread_fxn` and `video_thread_fxn` open the Codec Engine, using the Engine name passed from main.c to ensure that they open the same engine. Why is this done? (as opposed to opening the Codec Engine once in main.c and passing the handle to the audio and video threads)
   - Each thread must obtain its own engine handle using the Engine_open() function call to ensure conflicts do not arise between multiple threads attempting to access the Engine concurrently.

Lab 9.1

5. (Advanced) Show how you would modify the `inBufDesc` buffer descriptor building code of the `encode_audio` function to handle two input buffers instead of one as it is set up now (say for instance, a left and right channel). Use char *`inputBufferL` and char *`inputBufferR` as your two input buffers.
   - `inBufDesc.numbufs = 2;`
   - `inBufDesc.bufSizes = inBufSizeArray;`
   - `inBufDesc.bufs = inBufArray;`
   - `inBufSizeArray[0] = inBufSizeArray[1] = inputSize;`
   - `inBufArray[0] = inputBufferL;`
   - `inBufArray[1] = inputBufferR;`

6. (Advanced) Describe in your own words how the makefile incorporates the code contained in the published engine into the final application build.
   - The published engine is contained in an object file named `engine_debug_x470MV.o` or `engine_release_x470MV.o` depending on whether the release or debug version is required. The makefile simply links this object file in with the application object files using the `gcc` tool.
Lab 10

Lab 10 Questions

1. In the engine.cfg file, all codecs are specified with a local parameter of true. What does this imply regarding the need for a DSP server?
   That no DSP server is needed.

2. What differences are there between the runxdc.sh script used to build the application in lab10b_all_rtsc and the runxdc.sh script used to build the application in lab08b_audio_video_rtime?
   The XDCPATH environment variable has been modified to add the repository paths of those packages referenced in engine.cfg

3. What differences are there between the package.bld script used to build the application in lab10b_all_rtsc and the package.bld script used to build the application in lab08b_audio_video_rtime?
   The addExecutable function call has been modified to specify the engine.cfg configuration file in the parameters.

4. Not counting the VISA commands that need to be inserted into the application source files, what steps would be necessary to migrate the solution in lab08b_audio_video_rtime to the solution in lab10_all_rtsc?
   The above changes to runxdc.sh and package.bld in addition to creating an engine.cfg configuration file to configure the local engine.

Lab 11

Lab 11 Questions

1. How could engine.cfg be modified to use a DSP server image that is placed in a subdirectory of the directory into which the application executable is installed? (Say if the app is run from /opt/workshop, and the server is stored in /opt/workshop/servers.) If you wanted to use an absolute path, would this be an absolute path on the Host computer's linux filesystem, or an absolute path on the DaVinci board’s linux filesystem?
   
   demoEngine.server = "./server_debug.x64P";
   Changed to:
   demoEngine.server = "./servers/server_debug.x64P";
   or
   demoEngine.server = "/opt/workshop/servers/server_debug.x64P";

2. What would be the problem with the audio thread and the video thread using two different engines that use two different servers?
   When the Engine_open() call is made to an engine that uses a remote server, the server image is loaded onto the DSP and the DSP is taken out of reset. Since servers in the current Codec Engine Framework implementation are complete images, only one can run on the DSP at a time.
Lab 12

Lab 12 – Questions (1)

1. Which Operating System Abstraction Layer (OSAL) is specified in the server configuration file server.cfg? Which OSAL is specified in the engine configuration file? Why are these different?
   
   engine.cfg uses osalGlobal.DSPLINK_LINUX because it runs on Linux
   server.cfg uses osalGlobal.DSPLINK_BIOS because it runs on DSP/BIOS

2. What attributes are specified for the video and audio encoders and decoders when they are added into the server in the server configuration file? Which attributes are specified for the same codecs when added into the engine in the engine configuration file? Explain for each non-matching attribute why it makes sense for a codec added to an engine or server, but not vice-versa.
   
   engine: local, groupId
     (codecs on server are always local, groupId set by priority)
   server: stacksize, memID, priority
     (codecs on server run within their own threads, so these thread attributes are needed. Codecs on engine run within the thread context that the VISA call is made from)

Lab 12 – Questions (2)

3. All EDMA resources are shared between the Arm and DSP cores and must be allocated between them. For the lab12_build_server solution, how many QDMA channels are allocated for use by the DSP server based on the server.cfg file? How many param (parameter ram tables) are available for these QDMA channels to use? How many transfer complete codes have been allocated for use by the server?
   
   48 PARAMS, 8 QDMA channels and 8 tccs

4. (Advanced) We also see that the DMAN3 module (covered in more detail during the next chapter) has been configured in the server.cfg file. We can see from the configuration that when an algorithm requests memory of type DARAM0, this memory will come from the “L1DSRAM” memory segment. Where is this memory segment defined?
   
   The L1DSRAM memory segment is allocated in the server.tcf file, the BIOS configuration file.
Acronyms

API  Application Process Interface – defined protocol to interact with software components
APL  Application Layer – uppermost "master" layer in a DaVinci software system
ASP  Approved Software Provider – third party recognized by TI to provide DaVinci software support
A/V  Audio/Video – common multi-data grouping used in most video systems (sound and picture)
BIOS or "DSP/BIOS" - RTOS for TI DSPs
BSL  Board Support Library – extension of CSL for a given target board
BSP  Board Support Package – see LSP
CCS  Code Composer Studio – IDE for developing DSP software
CE  Codec Engine – IDE for developing DSP software
CGT  Code Generation Tools –
CSL  Chip Support Library – interface to peripherals via predefined structures and API
CXD  Create / Execute / Delete - three phases of the life cycle of a dynamic application
DIO  Device IO – interface between an IOM and SIO
DMA  Direct Memory Access – sub-processor that manages data copy from one place in memory to another
DMAN  Direct Memory Access Manager – TI authored framework to abstractly manage DMA channels
DSK  DSP Starter Kit – low-cost evaluation tool for developing DSP solutions
DSKT2 DSP Socket, rev 2 – TI authored framework to manage DSP memory allocations
DSP  Digital Signal Processor - processor with enhanced numerical abilities
DVEVM  Digital Video Evaluation Module – hardware platform allowing system test and application development
DVSDK  Digital Video Software Development Kit – adds DSP side development ability to DVEVM
EMIF  External Memory Interface – TI DSP sub-component that manages flow of data on/off chip
EPSI  Easy Peripheral Software Interface – DaVinci API for communication with IOL (drivers)
EVM  Evaluation Module – hardware test and debug platform
GHS  Green Hills Software – makers of Multi IDE and Integrity OS; TI ASP
GPP  General Purpose Processor – standard micro processor, as opposed to a special purpose processor
HLL  High Level Language – eg: C, C++, Java
HW  Hardware – physical components
HWAL  Hardware Adaptation Layer – TI software adapting DaVinci software framework to a given HW system
ICE  In Circuit Emulator – hardware debug tool
IDE  Integrated Development Environment –
IOM  Input Output Mini-driver – device driver standard in DSP/BIOS
IPC  Interprocessor Communication – interface between processors, eg: DSP/BIOS Link
IPO  Input / Process / Output – flow of data in a DSP system
JTAG  Joint Test Action Group – standard for interface to debug host
LSP  Linux Support Package – device/board specific kernel and driver software
OSAL  Operating System Adaptation Layer – TI software porting Codec Engine to a given GP OS
MAC  Multiply Accumulate – core activity in many DSP algorithms
McBSP  Multi Channel Buffered Serial Port – serial port that interfaces to numerous data formats
MIPS  Millions of Instructions Per Second – basic measure of DSP performance
MP3  MPEG 4 level 3 – video standard audio encoding methodology
MPEG  Motion Picture Experts Group – video compression standard
MPU  Micro Processor Unit – processor core
MV  MonteVista – Leading support provider for embedded Linux
MVL  MonteVista Linux – Version of Linux supported by TI for their DaVinci DM644x devices
OEM  Original Equipment Manufacturer – maker of a given hardware solution
OS  Operating System – software that provides sophisticated services to software authors
RAM  Random Access Memory – memory that can be read from or written to
RISC  Reduced Instruction Set Computer – processor with small, fast instruction set
RPC  Remote Procedure Call – calls that may invoke local or trans-processor functions
RTA  Real-Time Analysis – ability to observe activity on a processor without halting the device
RTSC  Real-Time System Component – new TI methodology for packaging software components
RTOS  Real-Time Operating System – software kernel that is tuned to deterministic temporal behavior
SARAM  Single Access RAM – RAM internal to the DSP cell that allows a transaction each cycle
SDRAM  Synchronous DRAM – clock driven (fast) Dynamic Random Access Memory
SPL  Signal Processing Layer – DaVinci software partition holding VISA Codecs and other DSP elements
SW  Software – code run on hardware
TI  Texas Instruments – semiconductor manufacturer specializing in DSP and analog
TTO  Texas Instruments Training Organization – group within TI chartered to provide DSP training
UART  Universal Asynchronous Receiver Transmitter – serial port with clock embedded in data
USB  Universal Serial Bus – modern interface between PCs, handhelds, and numerous peripherals
VISA  Video, Imaging, Speech, Audio – SPL API on DaVinci
xDAIS  eXpress DSP Algorithm Interface Standard – rules that define how to author DSP components
XDC  xDAIS for Digital Media – xDAIS extended to directly support VISA

Terms

3rd Party Network  TI approved list of vendors supporting DaVinci based system development
Codec  This term can be used in three ways. (1) Software engineers use this term to describe Compression/Decompression algorithms commonly used in video, imaging, speech, and audio applications. Examples include: MPEG2, H.264, JPEG, G.711, MP3, AAC. (2) In the world of DaVinci software, the term “codec” is often used to imply any signal processing algorithm. Is is common to hear algorithms that adhere to TI’s xDM or xDAIS software interface standard referred to as “codecs”. (3) From a hardware engineers point of view, the term codec refers to a single integrated circuit device which contains both an analogue-to-digital converter (ADC) as well as a digital-to-analogue converter (DAC). Note, most AV systems contain both types of codecs, which can be confusing when during consversations involving both software and hardware engineers.
Codec Engine  Software infrastructure developed by TI for use on DaVinci processors. Allows application code to leverage DSP algorithms without need to know the low level details
DevRocket  MontaVista’s Linux IDE
DSL Link  Physical transport layer for inter-processor communication
Engine  CE framework layer for managing local and remote function calls
Multi  GreenHills IDE
Server  Remote thread that services create/delete RPC’s from the engine
Skeleton  Function that unmarshals RPC arguments sent by stub
Stub  Function that marshalls RPC arguments for transport over DSP Link
Building Programs with the XDC Build Tool

Introduction

DaVinci software is built and packaged using the Express DSP Component (XDC) command line tool. This tool can create (and consume) Real Time Software Component (RTSC) packages. Similar, in many ways, to other make/build tools, XDC can build executable files from source files and libraries. Unlike other make/build tools, it can automatically perform dependency and version checking.

This chapter introduces the XDC tool and describes its basic features and requirements.

Learning Objectives

Objectives

- List the features of XDC and describe its benefits over other make tools
- Describe the three basic files required by XDC (package.xdc, package.bld, config.bld)
- Chart out the flow XDC follows when building a package and executables
- Build an application package using XDC
XDC Overview

XDC Build Overview (Ordinary)

- Like other build tools (i.e. gmake):
  - **XDC builds executables** or **libraries** according to a set of build instructions
  - XDC will **build any dependent files**. Of course, simple examples (like "hello.c") are not dependent on any other files.
  - XDC can build for **multiple targets** at once (e.g. ARM and DSP)
  - Sources can include C, C++, Assembly, Libraries & Archives

XDC Build Overview (ExtraOrdinary)

- **Unlike** other build tools, XDC **builds RTSC packages**
  - RTSC = Real Time Software Component
  - Packages can retain **dependency** and **version** information
  - When packages are used as sources, XDC **automatically performs dependency & version checking** (whereas gmake doesn't natively support it)
Moreover, unlike other build tools,
- **XDC can generate code** when provided with a configuration script. This is useful when combining package(s) into a new executable.
- This feature significantly **simplifies building complex executables** (like DaVinci Engines and DSP Servers) which will be explored later.

### Packages (and Repositories)

**XDC Packages**

- As mentioned earlier, **XDC creates** and can consume **packages**
- Library packages are often referred to as **smart libraries** or “smart archives”
- Along with program source and libraries, packages contain **metadata**:
  - **Version & Dependency** info is not explicitly used in this workshop, though these features are used within the component packages (e.g. Codec Engine) created by TI
  - **Module** information is discussed later in the workshop

Packages are described by a file ...
### package.xdc

**package.xdc**

- **Package Name**
- **Version Info**
- **Dependencies**
- **Modules**

```c
/*
* ======== application ========
* vendor: ti.tto
* app name: app
* pkg name: ti.tto.app
*/
package ti.tto.app {
}
```

- "package.xdc" describes a package:
  - Name, Dependencies, Version, lists its Modules
- Package name must be **unique**
- It's often common to name the package along these lines:
  - vendor . project name . name of what you're building (but don't use spaces)
- Packages are usually **delivered as an archive**: ti.tto.app.tar

---

**More package.xdc examples...**

---

### More “package.xdc” Examples

**Parts of package.xdc:**
- **Package Name**
- **Version Info**
- **Dependencies**
- **Modules**

```c
/*
* ======== codec ========
*/
requires ti.sdo.ce.audio [ver]
package codecs.auddec_copy [version] {
    module AUDDEC_COPY;
}
```

**Parts of package.xdc:**
- **Package Name**
- **Version Info**
- **Dependencies**
- **Modules**

```c
package app { }
```

---

**Where are packages located?**
Repositories

Directories containing packages are called **repositories**. For example, we could have two packages in the workshop/lab repository:

- **workshop**
- **lab**
- **ti.tto.app1.tar**
- **ti.tto.app2.tar**

In our workshop, we chose to use the same package name over and over: **app**
- Thus, we effectively chose to have a separate repository for each lab.
- While not necessarily common, this seemed the easiest way to handle a series of lab exercises and solutions.

**Repository (Example 1)**
- Repository: `/home/user/workshop/lab`
- Package name: **app**

- **workshop**
- **lab**
- **app**

**Misc. package files** (auto-generated by XDC)

**Package description files** (you'll supply these)

- **package.xdc**
- **package.bld**
- **hello.c**
- **hello.x470MV**
- **hello.x64P**

XDC creates a number of files and directories as part of a compiled package. While you must keep them, you will not need to work with them.
- Note, package.bld will be discussed later in this chapter.
### Package and Repository (Example 2)

**Example 2**
- **Repository:** `/home/user/workshop/lab`
- **Package names:** app, codecs
- **Delivered as:** app.tar, codecs.tar

### Package and Repository (Example 3)

**Example 3**
- **Repository:** `/home/user/dvevm/codec_engine/packages`
- **Package name:** ti.sdo.ce.video
- **Deliver:** ti.sdo.ce.video.tar.gz
Invoking XDC

XDC Invocation (Explicit Arguments)

- `XDC`<target files><XDCPATH><XDCBUILDCFG>

- Builds one or more files, as specified
- Files can also be specified via implicit commands (next slide)

- `XDCPATH` specifies the directories to be searched during build

- Specifies “config.bld” file which contains your platform's build instructions

Typing in all the search directories each time is tedious, so
Simplify XDC with scripts...

Invoking XDC with Shell Scripts (.SH)

```bash
#!/bin/sh
# Import installation paths
./setpaths.sh
# Define search paths
export XDCPATH=$DVEVM_INSTALL_DIR/packages; $SDK_INSTALL_DIR; $BIOS_INSTALL_DIR
# Define options for execution
export XDCBUILDCFG=$(pwd)/../config.bld
xc
```

```bash
#!/bin/sh
# Absolute Paths must be set per system ###
# DVEVM software directory
export DVEVM_INSTALL_DIR=
  "home/user/dvevm_1_10"
# Where Codec Engine package is installed
export CE_INSTALL_DIR=
  "$DVEVM_INSTALL_DIR/codec_engine_1_02"
# Installation directory of the software
# development kit (may be same as above)
export SDK_INSTALL_DIR="$DVEVM_INSTALL_DIR"
# where the BIOS tools are installed
export BIOS_INSTALL_DIR=
  "$DVEVM_INSTALL_DIR/bios_5_30"
...
export PATH="$XDC_INSTALL_DIR:$PATH"
```

XDC also has implicit args...
Invoking XDC

DaVinci DM644x Workshop - Building Programs with the XDC Build Tool
XDC Build Flow

**config.bld**

### XDC Flow : config.bld

<table>
<thead>
<tr>
<th>Invoke XDC</th>
<th>XDC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;files&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;XDCPATH&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;XDCBUILDCFG&gt;</td>
</tr>
</tbody>
</table>

- Platform Def's: `config.bld` (path specified by xdcbuildcfg)

**config.bld** defines platform-wide definitions

- Its path is defined by `XDCBUILDCFG` environment variable
- Usually copied from TI examples and edited once per platform
- This script is run prior to all build scripts. It sets host-system-independent values for targets and platforms, then it attempts to find the host-system-specific user.bld script that sets rootDirs.

Looking at `config.bld`...

Let’s start at the end of the file. It shows an array of build targets – each of which are defined at the top of the file.

### Parts of config.bld:

- **DSP Target**
- **ARM Target**
- **Linux Host Target**
- **Build Targets**

### config.bld

```bash
/*
 * ======== Build.targets ========
 * list of targets (ISAs & compilers)
 * you want to build for
 */
Build.targets = [
  C64P,
  MVArm9,
  Linux86
];
```

- First “platform” definition in `config.bld` is for the various targets which can be built when you run XDC
- You can add to this list, but probably won’t need to
- Notice that each target is defined at the top of the file (shown on the next 3 slides)
// ======== DSP target =============
/* Create Javascript var "remarks" - set equal to string of C6x C options to: enable remarks; warn if func proto not found */
var remarks = " -pden" + " -pdsw=225";

// Inherit target specified in TI provided target module package
var C64P = xdc.useModule("ti.targets.C64P");

// Modify elements of build target data structure
C64P.platform = "ti.platforms.evmDM6446";
C64P.ccOpts.prefix += " -k";
C64P.ccOpts.prefix += " --mem_model:data=far";
C64P.ccOpts.prefix += remarks;

// C64P codegen tool location ... C6000.CG set in setpaths.sh
C64P.rootDir = java.lang.System.getenv("C6000.CG");

- TI provides a default build models for each target (i.e. DSP, ARM, x86)
- useModule() is the XDC method to pull-in a RTSC package; ti.targets.C64P is a TI provided package
- Targets can be found at:
  /home/user/dvevm_1_10/xdctools_1_21/packages/ti/targets
- The defaults are inherited and overridden using config.bld (as shown above)

// ======== ARM target =============
var MVArm9 = xdc.useModule('gnu.targets.MVArm9');

MVArm9.ccOpts.prefix += " "
MVArm9.platform = "ti.platforms.evmDM6446";

/* add pthreads */
MVArm9.lnkOpts.suffix = "-lpthread " + MVArm9.lnkOpts.suffix;

/* MontaVista ARM9 tools path */
MVArm9.rootDir = java.lang.System.getenv("MONTAVISTA_DEVKIT") + "/arm/v5t_le/armv5tl-montavista-linuxeabi";
### Parts of config.bld:
- DSP Target
- ARM Target
- Linux Host Target
- Build Targets

```javascript
/*
  * ==== Linux host target ========
  */

var Linux86 = xdc.useModule('gnu.targets.Linux86');
Linux86.lnkOpts.suffix = "-lpthread " +
  Linux86.lnkOpts.suffix;
Linux86.rootDir = "usr";
```

### package.bld

**XDC Flow**

<table>
<thead>
<tr>
<th>Invoke XDC</th>
<th>XDC &lt;files&gt; &lt;XDCPATH&gt; &lt;XDCBUILDCFG&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Def's</td>
<td>→ config.bld (path specified by xdcbuildcfg)</td>
</tr>
<tr>
<td>Package Def's</td>
<td>→ package.xdc (current directory)</td>
</tr>
<tr>
<td>Executable/Library Build definitions</td>
<td>→ package.bld (current directory) addExecutable</td>
</tr>
</tbody>
</table>

- **package.bld** defines how a package should be built:
  - Sources to be built
  - Target executables to create
  - Build profiles (compiler/tool options)
- Written with javascript
- It may reference items defined in config.bld
- Must be located in directory where XDC is invoked (i.e. pkg path)
- Usually copied from TI examples and edited once per project
Build Package with Executable Program

```javascript
var Pkg = xdc.useModule('xdc.bld.PackageContents');
// Build package ... and Add an Executable to it
Pkg.addExecutable("hello_debug", MVArm9, ...);
```

Before Running XDC

```
```

After Running XDC

```
Looking more closely at addExecutable...

Pkg.addExecutable

```javascript
var Pkg = xdc.useModule('xdc.bld.PackageContents');
/*
* ======== Add Executable ========
*/
Pkg.addExecutable(
    "app_debug", // Name of executable file
    MVArm9, // Build target (from config.bld)
    MVArm9.platform, // Platform (from config.bld)
    {
        cfgScript: null, // Will use this in later chapters
        profile: "debug", // Build profile to be used
    }
).addObjects( ["main.c"] ); /* JavaScript array of obj files; if source files are passed, xdc will first build them into object files */
```

- The `addExecutable` method tells XDC what (and how) to build your executable program.
- This example is a "hard-coded" version. It needs to be edited for each lab.

Can we make `package.bld` more flexible?
### Generic package.bld

```javascript
var basename = "myApp";

// Define array of targets to build
var targs = [MVArm9, Linux86];

// Define array of profiles to build
var profiles = ["debug", "release"];

// ======== Add Executable ========
/* Cycles through the arrays of build targets and profiles 
and create an executable for each combination */
for (var i = 0; i < targs.length; i++) {
    for(var j = 0; j < profiles.length; j++){
        Pkg.addExecutable (
            basename + "." + profiles[j],
            targs[i],
            targs[i].platform,
            {
                cfgScript: null,
                profile: profiles[j]
            }
        ).addObjects ( ["main.c"] );
    }
}
```

- Use JavaScript to create a generic build routine
- Quickly change what's built by simply changing arrays
- Almost "One size fits all"

- Example on first pass:

```
Pkg.addExecutable ("myApp_debug", MVArm9, MVArm9.platform, 
{ cfgScript: null, profile: "debug" 
})
```

### Generic Source Array

```javascript
// ======== Source Array ========
// Generate an array of all source files in the directory
var sources = java.io.File('.').list();
var csources = [];
for (var i = 0; i < sources.length; i++){
    if(String(sources[i]).match(/.*\.c$/))
        csources.push(sources[i]);

// ======== Add Executable ========
for (var i = 0; i < targs.length; i++) {
    for(var j = 0; j < profiles.length; j++){
        Pkg.addExecutable {
            ... 
        ).addObjects ( csources );
    }
}
```

- Fancy JavaScript way to gather together all the source files found in the current directory (i.e. directory where package.bld resides)
- Simulates a common practice used in makefiles
- With our "Generic package.bld" and "Generic Source Array" we can use the same package.bld for all lab exercises
**package.bld**

**Parts of package.bld:**
- Define target array
- Define build profiles
- Source array
- Create executable(s)
- **Epilogue**

```c
/*
 * ======== Epilogue ========
 */

#include ../../custom.mak

install: 

...```

- Epilogue is not required
- Add additional *makefile* rules to the end of the file

---

**package.mak**

**XDC Flow**

<table>
<thead>
<tr>
<th>Invoke XDC</th>
<th>XDC &lt;files&gt; &lt;XDCPATH&gt; &lt;XDCBUILDCFG&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Def's</td>
<td>config.bld  (path specified by xdcbuildcfg)</td>
</tr>
<tr>
<td>Package Def's</td>
<td>package.xdc  (current directory)</td>
</tr>
<tr>
<td>Executable/Library Build definitions</td>
<td>package.bld  (current directory) addExecutable</td>
</tr>
<tr>
<td>Generated by XDC</td>
<td>package.mak  (internal XDC script to create)</td>
</tr>
</tbody>
</table>

- XDC generates and runs a file called package.mak
- You can open and examine package.mak, but you should not edit it (since it will be re-generated by XDC)
xDC basically performs two operations:
1. Makes executable outputs (similar, in fashion, to gmake)
2. Creates configuration code (we won’t use this feature till later)

Javascript .bld files provide build and configuration commands
- config.bld holds common build syntax
- Simultaneously build multiple targets
Lab 99 – Using XDC

Welcome to the compulsory “Hello World!” lab, where we will begin our exploration of the DaVinci Evaluation Module (DVEVM) and the software programming tools available to you in your development.

Lab Objectives

In this lab, you will:

- Build a simple application using the XDC build utility in conjunction with the shell scripting capability of a Linux host machine.
- Execute the “Hello world!” application on both the x86-based Linux host system and the Arm-926-based DaVinci target system using Linux terminals.

Workshop Lab Files

These files are common to all the lab exercises.

- **config.bld**: Specifies compiler, assembler and linker locations and options for three targets: Linux86, MVarm9 and C64P
- **setpaths.sh**: Specifies absolute paths that are referenced by runxdc.sh
- **runxdc.sh**: Invokes the xdc build tool, providing package search paths and the location of the config.bld script to use
- **main.c**: Application file containing main()
- **package.bld**: Xdc build script
- **package.xdc**: Xdc package declaration file

Lab99a_hello_world

1. Begin by logging into the Linux Host (i.e. desktop) Computer and opening a terminal by right clicking on the desktop and selecting “Open Terminal”

   You will begin in the /home/user directory (the home directory of the user named “user”), also represented by the tilde (~) symbol.
2. Descend into the /home/user/workshop directory using the “cd” command. (“cd” is short for “change directory”).

3. You may use the “ls” command to survey the contents of this directory. (Lower case “LS” is short for “list”)

   The workshop directory is the working directory for the lab exercises – and contains all the starter files needed for this workshop. (Note, the /home/user/solutions folder contains solutions for all of the workshop labs.) There are two additional files: config.bld and setpaths.sh. The config.bld file provides configuration information that is used by the XDC build tool. The setpaths.sh file contains absolute path listings for the locations of various tools and packages that will be used to build projects throughout the workshop.

   The setpaths.sh file contains the only absolute paths used in all of the workshop labs and solutions. The runxdc.sh scripts used in each lab to build the solutions reference the absolute paths set in setpaths.sh. As such, the workshop files can be placed in any directory within the Linux filesystem, and/or the various required packages and tools placed in any directory and still be made to build properly by changing only the paths set in setpaths.sh.

   For the DaVinci 4-day workshop, the proper filepaths have already been configured for the setup we are using. However, when you take your labs and solutions home to work on them further, you may need to modify the setpaths.sh in order to build correctly on your system.
4. Examine into the `lab99a_hello_world` directory.

   ```
   # cd ~/workshop/lab99a_hello_world
   ```

The `lab99a_hello_world` project has only one directory, “app.” (As our lab exercises become more complex, some projects will have multiple directories at this level.) The files which are used to build the `lab99a_hello_world` application are as follows (note that you may see additional files as the XDC tool generates files as well):

### Workshop Files (common to all lab exercises)
- `config.bld` – specifies compiler, assembler and linker locations and options for three targets: Linux86, MVArm9 and C64P
- `setpaths.sh` – specifies absolute paths that are referenced by `runxdc.sh`

### `lab99a_hello_world`
- `runxdc.sh` – invokes the XDC build tool, providing package search paths and the location of the `config.bld` script to use

### `app (found in lab99a_hello_world app directory)`
- `main.c` – application file containing the `main()` entry point
- `package.bld` – XDC build script
- `package.xdc` – XDC package declaration file

5. Examine the C source file `app/main.c` which prints the string “hello world” to standard output.

   ```
   # gedit app/main.c &
   ```

**Note:** You don’t need to type the pound sign “#”, this is shown to indicate the terminal prompt.

For help on using the `printf` command, you can access the man(ual) utility from within a Linux terminal by typing:

   ```
   # man printf
   ```

In this case, there is a `printf` terminal utility as well as a `printf.c` library command, so to get the page you want, you will need to type:

   ```
   # man -a printf
   ```

which will pull up all manual pages matching `printf`. Type “q” (no return key needed) to quit the current page and go to the next entry. The second page, `printf(3)`, is the page that you want. Note that the man page also shows the header file required to use this command.
6. **Examine app/package.xdc.**

   This is a very simple file that does nothing more than to declare the name of the package used to build the application. This name must match the directory that the package.xdc file resides in, so in this case the package name used is “app” (It could also have been “lab99a_hello_world.app” or even “workshop.lab99a_hello_world.app”).

   We will discuss packages in more detail in Chapter 10. For now we will simply state that the XDC tool requires a declared package in order to perform a build.

**Examine app/package.bld**

**package.bld** is the build script that specifies what executable(s) are to be built by the XDC tool and which files are used to build that executable. It is similar in function to a makefile script, just as the XDC tool itself is similar in function and purpose to the make utility.

7. **package.bld begins by declaring two arrays, one specifying all targets to be built, and one specifying all the profiles to be built. The script then declares a variable to store the base name of the executable you will generate.**

   While not a requirement, it is a good idea to build arrays that specify all targets you are building for and all profiles to build under so that you can quickly and easily add or remove a given target or profile. One great advantage to the XDC build tool is its capability to build for a number of targets (such as x86 host computer and arm-based DaVinci processor) and a number of profiles (such as debug and release) all simultaneously.

   Similarly the basename variable can be quickly modified. By placing these three variables at the top of the build script, most of the changes that will ever need to be made to this file are localized at the top.

8. **Next, package.bld iterates through the targets and profiles arrays described in step 7 in nested loops, and for every combination of target and profile specified, uses the Pkg.addExecutable method to create an executable. The Executable.addObjects() method, is used to add the “csources” array of C source files into the executable build.**

   You can refer to /home/user/dvsdk_1_30_00_40/doc/index.html for html-browsable links to XDC documentation, i.e.:

   ```
   # mozilla file: /home/user/dvsdk_1_30_00_40/xdctools_3_10/doc/index.html
   ```

   And from this page, select the XDC Object Model reference.

   Within the XDC Object Model reference, the Pkg.addExecutable method can be found under xdc.bld.PackageContents.addExecutable() (Pkg is an alias to xdc.bld.PackageContents), and the Executable.addObjects() method can be found under xdc.bld.Executable.addObjects()
**Examine runxdc.sh**

This is the shell script used to invoke the XDC tool. The shell script is used to set environment variables used by the tool, such as XDCPATH, the listing of search paths in which the tool will look for included packages, and XDCBLDCONFIG, the location of the config.bld script used to configure the tool.

9. **The script begins by executing the setpaths.sh script from the workshop top level.**

   The workshop places all required absolute path references into environment variables set in a single file called setpaths.sh. Each runxdc.sh script for each of the labs then references these environment variables. This allows the package path references for all of the labs to be updated by changing a single script (for instance if migrating to a new version of the Codec Engine or when being installed onto a new computer where various components have been installed in different locations.) While not required, this is often a helpful practice to implement.

   You should begin the runxdc.sh script for this lab by running the setpaths.sh script at the workshop top level. Recall that this command should be preceded by a single period indicating that the script will run in the current environment. Otherwise all of the environment variables set by the setpaths.sh script will be discarded when the script is exited. Using a relative path to the setpaths.sh script is preferable as it allows relocation of the workshop folder.

10. **Next, the script defines the XDCBLDCONFIG and XDCPATH environment variables using the export shell command.**

    (For instructions on using the export command, type “man export” in the Linux terminal)

    XDCBLDCONFIG and XDCPATH are two environment variables referenced by the XDC tool. For more information, refer to xdc.pdf, which is found at

    "# mozilla file: /home/user/dvsdk_1_30_00_40/xdctools_3_10/doc/index.html"

    This lab needs only to specify one XDC package path, “$CE_INSTALL_DIR/packages” (Where CE_INSTALL_DIR is the installation directory for the Codec Engine as specified in setpaths.sh) This is the path to the ti.platforms.evmDM6446 platform package module, which is the default platform for both the C64P and MVArm9 targets as set in config.bld. (*We’re not using the Codec Engine in this lab, it’s just that the platform description files referenced by config.bld are found in the same directory.*)

    The XDC build configuration file that the script specifies is config.bld at the top level of the workshop directory. It is best to use an absolute path here, so the script uses the $PWD environment variable to reference the directory that the runxdc.sh script is located within and then references config.bld relative to this location.

---

**Note:** PWD is an environment variable automatically maintained by Linux which contains the absolute path to the current directory. PWD stands for “print working directory”.

11. Finally, the script adds a line to execute the XDC command to make all packages in the current directory.

The script passes the $@ special variable as the first argument to XDC. $@ is a shell script symbol that references the array of parameters that were passed to the build script. By passing these directly to the XDC tool, we can run the script with different build goals such as “all”, “clean” and “install” Also, though it is not necessary for this simple single-package build, let’s go ahead and create a general script that will build all packages in the current directory by using the –P XDC option. You can use the asterisk (wildcard) character “*” in conjunction with the –P option to specify all subdirectories.

For more information on supported XDC environment variables and flags for the XDC tool, refer to xdc.pdf, which can be referenced at

# mozilla file: /home/user/dvsdk_1_30_00_40/xdctools_3_10/doc/index.html

**Build and Test the “Hello World” Application**

12. Build our **Hello World** application.

Run the runxdc.sh script in the Host RedHat Linux terminal.

```
# cd ~/workshop/lab99a_hello_world
# ./runxdc.sh
```

13. Run the x86-compiled version of the **Hello World** application.

The method for executing this file is the same as was used to execute the runxdc.sh script in step 12 of this procedure.

**Lab99b_extended_features**

The “hello world” project that we have will build arm and x86 executables just as desired, but we have not fully taken advantage of the features the JavaScript language provides for us.

First, it would be nice to build into the script a simple means of installing the DM6446 executable to a location where the DVEVM could run it (via a shared NFS folder).

Second, it would be nice to have our build script determine all source files in the current directory and automatically include them into the executable. This way, as we add files in the following labs, they will be included without needing to modify the script.

14. **Begin by copying the files from lab99a_hello_world into the lab99b_extended_features folder.**

```
# cp -R lab99a_hello_world/* lab99b_extended_features
```

The –R option indicates recursive copying. Lookup the cp man page for more info.

---

**Note:** There is an extra file, app/install.rule, already included in lab99b_extended_features, so be sure not to overwrite it.
Adding an install rule to the XDC build

As discussed in the lecture portion of this chapter, the XDC tool parses the `config.bld`, `package.xdc` and `package.bld` files associated with the project and uses them to create the file `package.mak`.

`package.mak` is a standard gnu makefile that is utilized to perform the actual build of any files specified in `package.bld`. Additionally, the XDC tool provides a mechanism for appending to the generated `package.mak` file using the `makeEpilogue` element of the Pkg object, for instance:

```makefile
Pkg.makeEpilogue = "This exact string will be added to the end of package.mak";
```

15. Cut and paste the code provided in the `lab99b_extended_features/app/install.rule` file to the end of `package.bld`.

Creating a string with the above text and appropriate variables substituted in is not difficult, but is mainly an exercise in string manipulation and tab and newline characters. For this reason, we have provided the code for you in a file named `install.rule` in the `lab99b_extended_features/app` directory.

What this code that you are cut and pasting into `package.bld` does is to append the following to `package.mak` (provided as pseudo-code below):

```
install: <basename>_profile1.x<suffix1> ...
 $(CP) $^ <exec_dir>

clean::
 $(RM) <exec_dir>/<basename>*
```

Where `<basename>`, `<profile>`, `<suffix>` are variables determined by the basename string, profiles array and targets array that are specified at the beginning of `package.bld`. By iterating through the profile and targets arrays, we can build all combinations of target and profile into the install rule’s dependencies and then copy these to `<exec_dir>`, which is a variable set in `config.bld` that specifies the directory (relative to the host’s filesystem) to copy executable files to so that they can be executed on the DVEVM. This is typically an NFS shared directory between the DVEVM and the host.

You can erase the `install.rule` file after you have copied it, if you like.

**Hint:** If you accidentally overwrote `install.rule`, just cut and paste this section from the `package.bld` of the solution file in `solutions/lab99b_extended_features/app`.

16. Test that the install rule is correctly functioning by changing directory to the top level of `lab99b_extended_features` and executing the `runxdc.sh` script with “install” as an argument.

```
# ./runxdc.sh install
```
17. Log a Linux terminal into the DVEVM if one is not already available.

The suggested terminal is via the serial connection. Within the Linux environment (i.e. within the VMWare virtual machine), the minicom utility is available. In the windows environment, the terra-term utility is available. Finally, for those more experienced with Linux, a telnet terminal can be used to connect to the board over Ethernet. All three of these methods have been configured in your setup.

18. Execute app.x470MV on the DaVinci EVM via the terminal opened in step 17.

The “install” phase of the build places this executable into the /opt/workshop directory on the board. (This corresponds to the /home/user/workdir/filesys/opt/workshop directory on the host as the board is booted using an NFS root mount that shares the /home/user/workdir/filesys directory from the host computer as its root directory.)
**Adding a source file search to package.bld**

Instead of manually specifying main.c as the source file for this build, we can take advantage of JavaScript’s java.io.File('<path>') object, which contains all of the files located at the given path. You can specify a path of '.' to indicate the current path and then use the list() method of this object to create an array of all files in the current path. Store this in an intermediate array such as “sources.”

Recall that the java.io.File('<path>') object contains all of the files located at a given path. You may then iterate through the “sources” array and use the String.match() standard JavaScript method of the String class to test each file in the directory against the following regular expression:

```
/.*\.c$/
```

For those files in the directory which match the regular expression, use the Array.push() method to place the matched file onto a new array called “csources”

---

**Hint: Regular Expressions**

**Hint:** While it is beyond this course to discuss regular expressions in detail, we will explain the above regular expression, hopefully providing a basis for those unfamiliar with regular expressions to potentially modify if needed.

**Hint:** The forward slashes (/) bracketing the expression are required to indicate that the text inside is a regular expression.

**Hint:** The period (.) indicates any character except for a newline.

**Hint:** The asterisk (*) indicates that the preceding set can be repeated any number of times, so a period followed by an asterisk (.* ) indicates any series of characters of any length.

**Hint:** The backslash (\) is an escape character, meaning that the character to follow it will be interpreted literally instead of with any special meaning assigned to that character. In this case, a backslash and period together (\. ) indicates simply the period as a character, instead of as a special character the way it was used previously.

**Hint:** The c is simply a character to be matched

**Hint:** Finally, the dollar sign ($) is a special character called an anchor and indicates that the preceding set must be matched at the end of the entry. In this case, it means “myFile.c” will be matched, but “myFile.cpp” will not be matched.

**Hint:** The expression above (/.*\.c$/), then, indicates <anything>.c
19. Instead of directly specifying the C source files for your build in an array called `csources`, use the `java.io.File().list()` method to search for all files in the current directory and the `String.match()` method to sort those files which have a `.c` extension.

This code should look like the following:

```javascript
var sources = java.io.File('.').list();
var csources = [];
for (var i = 0; i < sources.length; i++) {
    if(String(sources[i]).match(/.*\.c$/))
        csources.push(sources[i]);
}
```

Be careful of typos – it’s common to miss one of the “)”.

20. Finally, there is one last `package.mak` rule that we must add in using `Pkg.makeEpilogue` to include `../../custom.mak` (see discussion below).

Put the following statement at the end of `package.bld`.

```plaintext
Pkg.makeEpilogue += "include ../../custom.mak"
```

Recall that, as discussed in the lecture, the XDC tool parses the `config.bld`, `package.xdc` and `package.bld` project files and uses them to build `package.mak`, a gnu makefile that specifies how all files for the project are to be built.

As much as makefiles keep track of dependencies for files they build, the XDC tool keeps track of the dependencies for this `package.mak` file, which are `config.bld`, `package.xdc` and `package.bld`. As such, `package.mak` is only regenerated if one of these three files changes, which saves time in the build process.

However, since we are using JavaScript to build the `csources` array based on the contents of the current directory, the array of source files could change if a new source file is added into the directory or one is taken away. We therefore need to manually place a rule in `package.mak` that will rebuild itself if the contents of the current directory change. This is exactly what the code in `custom.mak` (the file we are including) does. Feel free to look through the file if you like, or simply use it as example code and include in any `package.bld` scripts that use the `java.io.File('..')` object.

21. Try building and running the project using the new `csources` code.
Questions

1. Which file in the lab99a_hello_world/app directory specifies that one or more executable files are to be built?

__________________________________________________________________________
__________________________________________________________________________

2. Which function call within this file actually scripts the building of a new executable file?

__________________________________________________________________________
__________________________________________________________________________

3. Examination of the config.bld file (at the top level of the workshop directory) shows that we have configured a third target, named C64P. This target is configured to build for the C64x+ DSP core that is available on the TMS320C6446. How could you modify the available files to build a Hello World executable that would run on the C64x+ core?

__________________________________________________________________________
__________________________________________________________________________

4. (Extended lab) Explain in your own words how the C source files used to build this application are specified in the package.bld build script of lab99b_extended_features.

__________________________________________________________________________
__________________________________________________________________________

5. (Extra credit) When compiling code for the C64x+ DSP, what compiler options are used for this lab exercise.

__________________________________________________________________________

Hint: It’s a combination of the C64x+ compiler strings from these files.

config.bld: ________________________________________________________________
ti.targets.C64P: ___________________________________________________________
package.bld: ____________________________________________________________
Challenges

Note: Before attempting any of the following challenges, please copy the
lab99b_extended_features folder into a new working folder, such as lab99_challenge1,
lab99_challenge2, etc. In order to copy the entire folder, you will need to use the “-R”
(recursive) flag with the “cp” command:

```
# cd /home/user/workshop
# mkdir lab99_challenge1
# cp -R lab99b_extended_features/* lab99_challenge1
```

Hint: The following man pages may come in handy
```
# man stdio.h
# man string.h
# man ctype.h
```

1. Modify the hello world application to ask the user’s name and use it in the greeting:

   ```
   # ./app_release.x470MV
   
   What is your name?
   Steve
   Hello, Steve!
   ```

2. Modify the hello world application to take the user’s name as a command line argument
   and use it in the greeting:

   ```
   # ./app_release.x470MV Steve
   
   Hello, Steve!
   ```

3. Modify the hello world application to determine the user’s name by either of the
   methods above. If the user’s name is “Steve”, greet as normal. Otherwise, display the
   message, “You’re not Steve!”:

   ```
   # ./app.x470MV Scott
   
   You’re not Steve!
   ```

4. (advanced) Modify the hello world application to determine the user’s name by either of
   the methods above, then convert it into an all-caps version in the greeting:

   ```
   # ./app.x470MV Steve
   
   Hello, STEVE!
Appendix 4 - H.264 Encoder Datasheet

Introduction

The codec datasheet is being included here for your convenience.
H.264 HD Baseline Profile Encoder (v01.41.00) on DM6446

FEATURES
- eXpressDSP™ Digital Media (XDM 0.9 IVIDENC) interface compliant
- Validated on the DM6446 EVM
- H.264 Baseline Profile upto level 4.1 supported
- Quarter-pel interpolation for motion estimation supported
- In-loop filtering, which can be switched off for whole picture and for slice boundaries supported
- User controllable multiple slices per picture supported
- Error-robustness features like intra slice insertion in inter frames, adaptive intra refresh, constrained intra prediction, and forcefully encoding of any frame, such as I frame supported
- User controllable quantization parameter range supported
- Unrestricted motion vector search, which allows motion vectors to be outside the frame boundary supported
- Image width and height that are non-multiple of 16 supported (multiples of 4, 8 supported, non-multiples of 4 not supported)
- TI proprietary rate control algorithms supported
- Arbitrary resolutions up to HD resolutions of 3840x2176 including standard image sizes such as PAL D1 (720x576), SQCIF, QCIF, CIF, QVGA, and VGA supported
- User configurable Group of Pictures (GOP) length supported
- User configurable parameters like pic_order_cnt_type, log2_max_frame_num_minus4, and chroma_qp_index_offset supported
- YUV422 interleaved and YUV420 planar color sub-sampling formats supported
- Controls the balance between encoder speed and quality by using the user defined motion estimation settings and encoding Preset option
- Constraint to keep macro block bits within 3200 bits as per the standard not supported
- Limited Video Usability Information (VUI) parameters support (timing information) in sequence parameter set information provided

DESCRIPTION
H.264 is the latest video compression standard from the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. This H.264 Encoder is validated on the DM6446 EVM with Code Composer Studio version 3.2.37.12 and code generation tools version 6.1.2.
Performance Summary
This section describes the performance of the H.264 HD Baseline Profile Encoder on DM6446 EVM.

Table 1. Configuration Table
<table>
<thead>
<tr>
<th>CONFIGURATION</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264 base profile levels up to level 4.1</td>
<td>H264_ENC_001</td>
</tr>
</tbody>
</table>

Table 2. Cycles Information - Profiled on DM6446 EVM with IMCOP and Code Generation Tools Version 6.1.2

<table>
<thead>
<tr>
<th>CONFIGURATION ID</th>
<th>PERFORMANCE STATISTICS (MEGA CYCLES PER SECOND)(^{(1)(2)(3)})</th>
<th>AVERAGE(^{(4)})</th>
<th>PEAK(^{(5)(6)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>H264_ENC_001</td>
<td>720p5994_parkrun_ter_1280x720_422i_205fr.yuv, targetFrameRate=13fps, targetBitRate=2Mbps, IntraPeriod=0.5sec, encodingFrameRate=16fps, rcAlgo=PLR4_RC, NFAVG=102</td>
<td>516</td>
<td>541</td>
</tr>
<tr>
<td></td>
<td>smaninrest_p1920x1080_30fps_420pl_60fr, targetFrameRate=5fps, targetBitRate=9Mbps, IntraPeriod=1sec (encoding Frame rate=5fps), rcAlgo=PLR4_RC, NFAVG=10</td>
<td>578</td>
<td>627</td>
</tr>
<tr>
<td></td>
<td>mobile_p352x288_30fps_420pl_300fr, targetFrameRate=30fps, targetBitRate=0.768Mbps, IntraPeriod=1sec (encoding Frame rate=30fps), rcAlgo=PLR4_RC, NFAVG=100</td>
<td>139</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>sheilds_720x480_422i_252frames.yuv, targetFrameRate=30fps, targetBitRate=1.572Mbps, IntraPeriod=0.5sec (encoding Frame rate=30fps), rcAlgo=PLR4_RC, NFAVG=252</td>
<td>421</td>
<td>452</td>
</tr>
<tr>
<td></td>
<td>Amazing_caves_1280x720_yuv422_250frames targetFrameRate=13fps, targetBitRate=3Mbps, IntraPeriod=0.5sec (encoding Frame rate=15fps), rcAlgo=PLR4, NFAVG=125</td>
<td>569</td>
<td>621</td>
</tr>
<tr>
<td></td>
<td>tennis_p704x480_30fps_420pl_150fr, targetFrameRate=30fps, targetBitRate=1.572Mbps, IntraPeriod=1sec (encoding Frame rate=30fps) rcAlgo=PLR4_RC, NFAVG=150, IntraPeriod=0.5 sec (encoding Frame rate=15fps)</td>
<td>513</td>
<td>537</td>
</tr>
<tr>
<td></td>
<td>720p50_shields_ter_1280x720_422i_300fr.yuv, target frame rate = 13fps, targetBitRate=2Mbps,rcAlgo=PLR4_RC, NFAVG=150,IntraPeriod=0.5 sec (encoding Frame rate=15fps)</td>
<td>345(^{(7)})</td>
<td>385(^{(7)})</td>
</tr>
<tr>
<td></td>
<td>720p5994_parkrun_ter_1280x720_422i_205fr.yuv, targetFrameRate=13fps, targetBitRate=3Mbps, IntraPeriod=1sec (encoding Frame rate=15fps), rcAlgo=PLR4_RC, NFAVG=102</td>
<td>448(^{(7)})</td>
<td>475(^{(7)})</td>
</tr>
</tbody>
</table>

\(^{(1)}\) Measured with program memory, stack, and I/O buffers in external memory with cache configuration : 32 K-bytes L1P Program Cache, 64 K-bytes L1D Data Memory, and 16 K-bytes L1D Data Cache, 64 K-bytes L2 Cache, 32 bit DDR @ 162 MHz, CPU @ 594 MHz and only used by encoder.
\(^{(2)}\) For random noise input, the cycles consumption of encoder increases by 10 to 15%.
\(^{(3)}\) Average and peak MCPS measurements can vary by +/-5%.
\(^{(4)}\) Average MCPS is calculated by multiplying average Mega Cycles Per Frame numbers by target frame rates. Average MCPS is calculated over NFAVG frames.
\(^{(5)}\) Peak MCPS is calculated on moving average of 4 frames over NFAVG frames.
\(^{(6)}\) For higher bit-rates of encoding, actual cycles performance will be worse than the above mentioned numbers.
\(^{(7)}\) For these, encodingPreset = High speed, option is used. All other performance numbers use (encodingPreset = User_defined, which uses High quality option).
Figure 1. Encoding Time for Individual Frames (720p50_shields_ter_1280x720_422i_300fr.yuv @ 2Mbps @ 15 fps, with 1 MV, QPI, LPF, UMV. Target Frame rate = 13fps, rc algo PLR4_RC,NFAVG = 150 Intra Period = 0.5sec and User_defined Preset)

Table 3. Memory Statistics - Generated with Code Generation Tools Version 6.1.2

<table>
<thead>
<tr>
<th>CONFIGURATION ID</th>
<th>LEVEL AND RESOLUTION</th>
<th>MEMORY STATISTICS(1)</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PROGRAM MEMORY</td>
<td>DATA MEMORY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTERNAL</td>
<td>EXTERNAL</td>
</tr>
<tr>
<td>H264_ENC_001</td>
<td>Level 3.0 D1 (720x576)</td>
<td>163</td>
<td>63.25</td>
</tr>
<tr>
<td></td>
<td>Level 4.0 HD (1280x720)</td>
<td>163</td>
<td>63.25</td>
</tr>
<tr>
<td></td>
<td>Level 4.0 HD (1920x1080)</td>
<td>163</td>
<td>63.25</td>
</tr>
<tr>
<td></td>
<td>Level 4.1 HD (3840x2176)</td>
<td>163</td>
<td>63.25</td>
</tr>
</tbody>
</table>

(1) All memory requirements are expressed in kilobytes (1 K-byte = 1024 bytes) and there could be a variation of around 1-2% in numbers.

Table 4. Internal Data Memory Split-Up

<table>
<thead>
<tr>
<th>CONFIGURATION ID</th>
<th>DATA MEMORY - INTERNAL (1)</th>
<th>INSTANCE (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHARED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONSTANS</td>
<td>SCRATCH</td>
</tr>
<tr>
<td>H264_ENC_001</td>
<td>0</td>
<td>63.25</td>
</tr>
</tbody>
</table>

(1) Internal memory refers to L1DRAM. All memory requirements are expressed in kilobytes and there could be a variation of around 1-2% in numbers.

(2) I/O buffers not included. Some of the instance memory buffers could be scratch.

Table 5. Co-Processor(s) Memory Statistics

<table>
<thead>
<tr>
<th>CONFIGURATION ID</th>
<th>SEQ DATA MEMORY (1)</th>
<th>SEQ PROG MEMORY (1)</th>
<th>IMX WORKING MEM (1)</th>
<th>IMX IMG BUF (1)</th>
<th>IMX CMD MEM (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H264_ENC_001</td>
<td>1</td>
<td>4</td>
<td>27</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

(1) All memory requirements are expressed in kilobytes and all are scratch buffers.
### Table 6. PSNR and Bit-Rate Details

<table>
<thead>
<tr>
<th>TEST SEQUENCE</th>
<th>BITRATE / AVERAGE LUMA PSNR (in dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1572000 (bps)</td>
</tr>
<tr>
<td>parkrun_p1280x720_30fps_420pl_302fr, target frame rate = 7.5 fps, rcAlgo=PLR3_RC, NFAVG=75, IntraPeriod=1 sec</td>
<td>-</td>
</tr>
<tr>
<td>shields_p1280x720_30fps_420pl_302fr, target frame rate = 7.5 fps rcAlgo=PLR3_RC, NFAVG=75 IntraPeriod=1 sec</td>
<td>-</td>
</tr>
<tr>
<td>stockholm_p1280x720_30fps_420pl_302fr, target frame rate = 7.5 fps rcAlgo=PLR3_RC, NFAVG=75 IntraPeriod=1 sec</td>
<td>-</td>
</tr>
<tr>
<td>tennis_p704x480_30fps_420pl_150fr.yuv, target frame rate = 30 fps rcAlgo=DCES_TM5, NFAVG=100 IntraPeriod=0.5 sec</td>
<td>30.3</td>
</tr>
<tr>
<td>stockholm_p1280x720_30fps_420pl_302fr, target frame rate = 7.5 fps rcAlgo=DCES_TM5, NFAVG=150 IntraPeriod=0.5 sec</td>
<td>-</td>
</tr>
</tbody>
</table>

Note:
1. Scene change detection is OFF
2. Loop Filter enabled
Notes

- I/O buffers:
  - Input buffer size = 4 M-bytes (1080P, one YUV422 interleaved frame)
  - Output buffer size = 1920*1080*2 = 4050k bytes (for encoding one 1080P frame)

- Memory Configuration
  - L1P : 32 K-bytes Program Cache
  - L1D : 64 K-bytes Data Memory and 16K-bytes Data Cache
  - L2 : 64 K-bytes Cache

- The performances obtained in Table 2 are sensitive to algorithm code placement. Refer the sample linker file provided in the test application setup for algorithm code placement. This is used for profiling in Table 2.

- The algorithm uses 6 QDMA channels and parameter space equal to 35 parameter entries. The algorithm uses DMAN3 interface for logical allocation of these channels.

- Total data memory for N non-pre-emptive instances = Constants + Runtime Tables + Scratch + N * (Instance + I/O buffers + Stack)
- Total data memory for N pre-emptive instances = Constants + Runtime Tables + N * (Instance + I/O buffers + Stack + Scratch)

References

- ISO/IEC 14496-10:2005 Information technology -- Coding of audio-visual objects -- Part 10: Advanced Video Coding
- H264 HD Baseline Profile Encoder on DM6446 User’s Guide (literature number: SPRUFBQ6B)

Glossary

<table>
<thead>
<tr>
<th>TERM</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>Elements that go into .const memory section</td>
</tr>
<tr>
<td>Scratch</td>
<td>Memory space that can be reused across different instances of the algorithm</td>
</tr>
<tr>
<td>Shared</td>
<td>Sum of Constants and Scratch</td>
</tr>
<tr>
<td>Instance</td>
<td>Persistent-memory that contains persistent information - allocated for each instance of the algorithm</td>
</tr>
</tbody>
</table>

Acronyms

<table>
<thead>
<tr>
<th>ACRONYM</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIF</td>
<td>Common Intermediate Format</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DMAN3</td>
<td>DMA Manager</td>
</tr>
<tr>
<td>EVM</td>
<td>Evaluation Module</td>
</tr>
<tr>
<td>GOP</td>
<td>Group of Pictures</td>
</tr>
<tr>
<td>IDR</td>
<td>Instantaneous Decoding Refresh</td>
</tr>
<tr>
<td>LPF</td>
<td>Loop Filter</td>
</tr>
<tr>
<td>MV</td>
<td>Motion Vector</td>
</tr>
<tr>
<td>QCIF</td>
<td>Quarter Common Intermediate Format</td>
</tr>
<tr>
<td>QDMA</td>
<td>Quick Direct Memory Access</td>
</tr>
<tr>
<td>QPI</td>
<td>Quarter Pel Interpolation</td>
</tr>
<tr>
<td>QVGA</td>
<td>Quarter Video Graphics Array</td>
</tr>
<tr>
<td>SQCIF</td>
<td>Sub Quarter Common Intermediate Format</td>
</tr>
<tr>
<td>UMV</td>
<td>Unrestricted Motion Vectors</td>
</tr>
<tr>
<td>VGA</td>
<td>Video Graphics Array (640x480 resolution)</td>
</tr>
<tr>
<td>XDM</td>
<td>eXpressDSP Digital Media</td>
</tr>
</tbody>
</table>
Revision History

This data sheet revision history highlights the changes made to the SPRS527A codec specific data sheet to make it SPRS527B.

Table 7. Revision History for H264 HD Encoder on DM6446

<table>
<thead>
<tr>
<th>SECTION</th>
<th>ADDITIONS/MODIFICATIONS/DELETIONS</th>
</tr>
</thead>
</table>
| Section 1 | Features:  
• Added the feature ‘Limited Video Usability Information (VUI) parameters support (timing information) in sequence parameter set information provided’ |
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or “enhanced plastic.” Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DLP® Products</td>
<td>Broadband</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Control</td>
</tr>
<tr>
<td>Clocks and Timers</td>
<td>Medical</td>
</tr>
<tr>
<td>Interface</td>
<td>Military</td>
</tr>
<tr>
<td>Logic</td>
<td>Optical Networking</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Security</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Telephony</td>
</tr>
<tr>
<td>RFID</td>
<td><a href="http://www.ti.com/telephony">www.ti.com/telephony</a></td>
</tr>
<tr>
<td>RF/IF and ZigBee® Solutions</td>
<td>Wireless</td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2009, Texas Instruments Incorporated