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 - 2011 by Texas Instruments Incorporated.
All rights reserved.

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

Revision History

October 2006, Version 0.80 (alpha)
December 2006, Versions 0.90/0.91 (alpha2)
January 2007, Version 0.92 (beta)
February 2007, Version 0.95
March 2008, Versions 0.96 (errata)
April 2008, Version 0.98 (chapter rewrites & errata)
September 2008, Version 1.30 (beta 1 & 2)
October 2008, Version 1.30 (beta 3)
February 2010, Version 2.00
August 2010, Version 2.10
Oct 2010, v3.03; Jan 2011, v3.05; Apr 2011, v3.06a; July 2011, v3.07
Aug 2011, Version 3.08
Welcome Topics

Workshop Goals
Where To Get Additional Information/Training?
Workshop Outline
About You
Administrative Topics
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: EVM, SDK/DVSDK

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

So, the focus of the workshop is...

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>Building Linux based Systems (ARM or ARM-DSP processors)</th>
<th>DaVinci / OMAP / Sitara System Integration Workshop using Linux (4-days)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="http://www.ti.com/training">www.ti.com/training</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Building BIOS based Systems (DSP processors)</th>
<th>System Integration Workshop using DSP/BIOS (4-days)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="http://www.ti.com/training">www.ti.com/training</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Developing Algo’s for C6x DSP’s (Are you writing/optimizing algorithms for latest C64x+ or C674x DSP’s CPUs)</th>
<th>C6000 Optimization Workshop (4-days)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="http://www.ti.com/training">www.ti.com/training</a></td>
</tr>
</tbody>
</table>

Online Resources:

- DaVinci Open-Source Linux Mail List

- Gstreamer and other projects

- TI Software
  - http://www.ti.com/myregisteredsoftware

- This workshop presentation & exercises

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

A few references, to get you started:

Non-TI Curriculum

<table>
<thead>
<tr>
<th>Linux</th>
<th>&quot;Linux For Dummies&quot;, by Dee-Ann LeBlanc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;Linux Pocket Guide&quot;, by Daniel J. Barrett</td>
</tr>
<tr>
<td></td>
<td>free-electrons.com/training</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.linux-tutorial.info/index.php">www.linux-tutorial.info/index.php</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.oreilly.com/pub/topic/linux">www.oreilly.com/pub/topic/linux</a></td>
</tr>
<tr>
<td></td>
<td>The Linux Documentation Project: <a href="http://www.tldp.org">www.tldp.org</a></td>
</tr>
<tr>
<td></td>
<td>Rute Linux Tutorial: <a href="http://rute.2038bug.com/index.html.gz">http://rute.2038bug.com/index.html.gz</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedded Linux</th>
<th>&quot;Building Embedded Linux Systems&quot;, by Karim Yaghmour</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;Embedded Linux Primer&quot;, by Christopher Hallinan</td>
</tr>
<tr>
<td></td>
<td>free-electrons.com/training</td>
</tr>
</tbody>
</table>


| ARM Programming (not required for Linux based designs)              | http://www.arm.com/ |

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="http://lwn.net/Kerne/LLD3/">http://lwn.net/Kerne/LLD3/</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.adeneo.com">www.adeneo.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Video</th>
<th>&quot;The Art of Digital Video&quot;, John Watkinson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;Digital Television&quot;, H. Benoit</td>
</tr>
<tr>
<td></td>
<td>&quot;Video Demystified&quot;, Keith Jack</td>
</tr>
<tr>
<td></td>
<td>&quot;Video Compression Demystified&quot;, 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
- 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 GMAKE (and Configuro)
- 6. Given: File Audio; Build: Audio In Audio Out
- 7. Setup an On-Screen Display (scrolling banner)
  - Video In Video Out
- 8. Concurrently run audio and video loop-thru programs

#### Using the Codec Engine
- 9. Use a provided Engine (containing local codecs)
- 10. Build an Engine (given local codecs)
- 11. Use remote codecs (using a provided DSP Server)
  - Swap out video codec for real H.264 codec
- 12. Build a DSP Server (given DSP-based codecs)
  - Use dot-product routine from C6Accel

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

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
TI ARM / ARM+DSP Devices

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 ARM and ARM+DSP 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 criteria
# Chapter Topics

| TI ARM / ARM+DSP Devices_yaml
|---|
| TI Embedded Processors Portfolio_yaml | 1-3
| System Examples_yaml | 1-4
| What Processing Do You Need?_yaml | 1-5
| ARM Core_yaml | 1-6
| DSP Core_yaml | 1-7
| Accelerator : 3D Graphics_yaml | 1-11
| Accelerator : Audio/Video Algorithms_yaml | 1-12
| Accelerators : Video Port (VPSS) / Display (DSS)_yaml | 1-14
| Peripherals_yaml | 1-19
| PRU – Programmable Real-time Unit_yaml | 1-20
| Moving Data Around – SCR / EDMA3_yaml | 1-21
| Final Considerations_yaml | 1-22
| Memory Map & MMU_yaml | 1-22
| Access to Peripherals_yaml | 1-24
| Pin Muxing_yaml | 1-24
| Choosing a Device : Web Tool_yaml | 1-25
## TI Embedded Processors Portfolio

### Microcontrollers

<table>
<thead>
<tr>
<th>16-bit</th>
<th>32-bit Real-time</th>
<th>32-bit ARM</th>
<th>ARM+</th>
<th>ARM + DSP</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430</td>
<td>C2000™</td>
<td>ARM</td>
<td>ARM+</td>
<td>ARM + DSP</td>
<td>DSP</td>
</tr>
<tr>
<td>Ultra-Low Power</td>
<td>Fixed-Point</td>
<td>Industry-Std</td>
<td>High-End DSP</td>
<td>Leadership DSP</td>
<td>Performance</td>
</tr>
<tr>
<td>Up to 25 MHz</td>
<td>Up to 300 MHz</td>
<td>&lt;100 MHz</td>
<td>4800 MMAC/ 1.07 DMIPS/MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash 1 KB to 256 KB</td>
<td>Flash 32 KB to 512 KB</td>
<td>USB, ENET, ADC</td>
<td>MMU, Cache</td>
<td>34,000 MMACs</td>
<td></td>
</tr>
<tr>
<td>Analog, ADC, LCD, USB, RF, Measurement, Sensing, General Purpose</td>
<td>Motor Control, Digital Power, Lighting, Sensing</td>
<td>Host Control</td>
<td>ARM + Cortex A-8</td>
<td>Up to 3 MB</td>
<td></td>
</tr>
<tr>
<td>$0.49 to $9.99</td>
<td>$1.50 to $26.00</td>
<td>$2.99 to $26.00</td>
<td>$5.00 to $25.00</td>
<td>$12.99 to $99.99</td>
<td>$1.99 to $99.99</td>
</tr>
</tbody>
</table>

### ARM-Based

<table>
<thead>
<tr>
<th>ARM+</th>
<th>ARM + DSP</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM9 Cortex A-8</td>
<td>Industry-Std Core</td>
<td>Industry-Std Core + DSP for Signal Processing</td>
</tr>
<tr>
<td>Industry-High Core</td>
<td>MMU, Cache</td>
<td>Leadership DSP Performance</td>
</tr>
<tr>
<td>High-End DSP</td>
<td>34,000 MMACs</td>
<td>34,000 MMACs</td>
</tr>
</tbody>
</table>

### DSP

- Leaders in DSP Performance
- 34,000 MMACs
System Examples

[Diagram of a set-top box (STB) with various components and connections, including Ethernet, Video Output, Digital Audio, Analog Audio, RS-232 Control, etc.]

[Diagram of a digital camera with components labeled: DDR2, CCD/CMDS, Power Management (TPS65020), Storage (SD/MMC), Connectivity (USB 2.0 or RS232), Video (TV Out NTSC/PAL), and Buttons & Feedback (GPIO or I2C).]
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?

- User Controls, GUI, OSD
- Peripheral Drivers
- Ethernet (other system comm)

DSP/BIOS™
- Video processing decoding, encoding, etc.
- Audio processing decoding, encoding, etc.

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 (375M to 1GHz) drive complex applications running on Linux, WinCE or Android systems

3D 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 multimedia processing expected by users of today's end products
- Think of the DSP as the ultimate, programmable hardware accelerator
- Video Accelerators – either stand-alone or combined with the DSP provide today's meet today's video demands with the least power required

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

NOTE
Features not available on all devices

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

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
What Processing Do You Need?

ARM Core

TI ARM CPU Processor Cores

- Classic ARM Processors
- Embedded Cortex Processors
- Application Cortex Processors
- Cortex-A8
- ARM11
- Cortex-R4
- ARM9
- Cortex-M4
- Cortex-M3
- ARM7

Capability

Performance, Functionality


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

ARM Processors : ARM+DSP

- Classic ARM Processors
- Embedded Cortex Processors
- Application Cortex Processors
- Cortex-A8
- ARM9
- Lower Cost

- 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 – 1.5 GHz) is ideal for high compute and graphic intense applications

**TI ARM Devices**

<table>
<thead>
<tr>
<th>ARM</th>
<th>ARM+DSP</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM1705</td>
<td>OMAP-L137</td>
<td>C6747</td>
</tr>
<tr>
<td>AM1707</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM1806</td>
<td>OMAP-L138</td>
<td>C6748</td>
</tr>
<tr>
<td>AM1808</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM355</td>
<td>DM644x</td>
<td>DM6437</td>
</tr>
<tr>
<td>DM365</td>
<td>DM6467</td>
<td></td>
</tr>
<tr>
<td>OMAP3503</td>
<td>OMAP3525</td>
<td></td>
</tr>
<tr>
<td>OMAP3515</td>
<td>OMAP3530</td>
<td></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>DM3725</td>
<td></td>
</tr>
<tr>
<td>AM3715</td>
<td>DM3730</td>
<td></td>
</tr>
</tbody>
</table>

**PIN-for-PIN Compatibility**

**DSP Core**

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

- Get Data
- DSP (Do Data)
- Put Data

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:

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

for (i = 1; i < count; i++){
    \( Y += \text{coeff}[i] \cdot x[i]; \)
What Processing Do You Need?

- ‘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

C64x
- Video/Imaging
- Enhanced EDMA

C62x
- C621x
- C671x
- Floating Point

C64x+
- L1 RAM/Cache
- Compact Instr’s
- EDMA3

C67x+
- 1 GHz
- EDMA (v2)
- 2x Register Set
- SIMD Instr’s

C671x

C67x
- EDMA
- L1 Cache
- L2 Cache/RAM
- Lower Cost

Available on the most recent releases

C674
- Fixed and Floating Point
- Up to 1.5 GHz
- Lower power
- EDMA3
- PRU

C66x
- 1.2 GHz
- EDMA3
- SPLOOP
- 32x32 Int Multiply
- Enhanced Instr for FIR/FFT/Complex

C64x+
- L1 RAM and/or Cache
- Timestamp Counter
- Compact Instr’s
- Exceptions
- Supervisor/User modes

C64x
- Combined Instr Sets from C64x+/C67x+
- Incr Floating-pt MHz
- Lower power
- EDMA3
- PRU
- Up to 1.5 GHz

C621x
- DMAX (PRU)
- 2x Register Set
- FFT enhancements

C62x
- C67x
- C671x

Linux Embedded System Design Workshop - TI ARM / ARM+DSP Devices
What Processing Do You Need?

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
What Processing Do You Need?

Accelerator: 3D Graphics

**Video ≠ Graphics**

- **Video**
  - Captured
  - Motion
    - Integrated image
    - Base image + motion
  - YUV/YCbCr format
    - Convert to RGB for display
  - Video codec: block-based
    - MPEG, WMV, JPG, etc.
  - Natural attributes
    - Digitized analog image

- **Graphics**
  - Generated
  - Animation
    - Multi-layered image
    - New delta images
  - RGB format
    - Ready for display
  - Graphics: pixel-based
    - BMP, GIF, etc.
  - Calculated attributes
    - Synthesized digital image

- 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.

**OMAP35x: ARM+DSP+Graphics**

- 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:
- AM3513
- AM3517
- DM3715
- DM3730

Texas Instruments
What Processing Do You Need?

Accelerator: Audio/Video Algorithms

**DM35x/DM36x**

![TMS320DM365 digital media processor](image)

**DM6446 - VICP**

**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
    - VC1/WMV9 D1 Encode
    - MPEG2 MP D1 Decode
    - MPEG4 ASP D1 Decode
    - Etc.
**What Processing Do You Need?**

**OMAP35xx – IVA2.2**

**OMAP35x Processors: ARM+DSP+Graphics**

<table>
<thead>
<tr>
<th>OMAP3525</th>
<th>OMAP3530</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM® Cortex™ A8</td>
<td>ARM® Cortex™ A8</td>
</tr>
<tr>
<td>C64x+ DSP &amp; video accelerator</td>
<td>C64x+ DSP &amp; video accelerator</td>
</tr>
<tr>
<td>L2 256kB</td>
<td>L2 256kB</td>
</tr>
<tr>
<td>L1P 16kB</td>
<td>L1P 16kB</td>
</tr>
<tr>
<td>L1D 16kB</td>
<td>L1D 16kB</td>
</tr>
<tr>
<td>ADL 100MHz</td>
<td>ADL 100MHz</td>
</tr>
<tr>
<td>Pin-for-pin compatible</td>
<td>Pin-for-pin compatible</td>
</tr>
</tbody>
</table>

Oct 2005
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)
What Processing Do You Need?

Front End: Resizer, Previewer, 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)

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

Video Window 0
- Video Window 1
  - OSD Window 0 (bitmap)
  - OSD Window 1 (bitmap)
  - Cursor

/dev/fb/1
/dev/fb/3
/dev/fb/0
/dev/fb/2
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

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
What Processing Do You Need?

**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
  - LCD Output Path
  - TV Output Path

**OMAP (DSS) Overlay Example**

- Background
- Video 2
- Video 1
- Graphics

- Video
- Audio
- Extras

- Week 15
- 8:52 am

*Texas Instruments*
What Processing Do You Need?

OMAP (DSS) Overlay Manager
Color Key Example

Top Layer
(Video 1 or 2)

Source
Transparency
Color Key

Bottom Layer
(Graphics)

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
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 add'l 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
## Final Considerations

### Memory Map & MMU

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

**DM6446 Memory Map**

- **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

**DSP linker segments**

- 120M  
  - CMEM
- 122M  
  - DDR heap (DSP heap)
- 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

### TMS320DM6446 Peripheral Ownership

<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 mux utility...
Final Considerations

**Choosing a Device: Web Tool**

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

[Web Tool Image]

[Web Tool Description]

[Website Link]

To force proper pagination, this page was intentionally left almost blank.
# Intro to TI's Foundation Software

## Introduction

At this point we have seen an overview of TI’s ARM and ARM+DSP devices (Sitara, DaVinci, OMAP, Integra). Next we will explore the basic components of the TI 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>♦ Algorithms/Codecs (Signal Processing)</th>
<th>xDAIS (eXpressDSP™ Algorithm Standard)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xDM (extending xDAIS with Classes)</td>
</tr>
<tr>
<td></td>
<td>VISA vs xDM</td>
</tr>
<tr>
<td></td>
<td>Where to Get Your Algorithms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>♦ Codec Engine Details</th>
<th>ARM or ARM+DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RPC - Remote Procedure Call</td>
</tr>
<tr>
<td></td>
<td>Code Review</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>♦ Software Summary</th>
<th></th>
</tr>
</thead>
</table>
Chapter Topics

Intro to TI's Foundation Software .................................................. 2-1

Application Layer ........................................................................ 2-3
Concepts .................................................................................... 2-3
I/O Layer - Linux Device Drivers .................................................. 2-6
Access Signal Processing Horsepower using the Codec Engine’s VISA Interface .................................................. 2-7
Adding VISA to our “Master Thread” ............................................. 2-9

Signal Processing Layer – Algorithms and Codecs .......................................................... 2-10
What is a “Codec” ........................................................................ 2-10
xDAIS (eXpressDSP™ Algorithm Standard) ........................................ 2-11
xDM (Extending xDAIS with Algorithm Classes) ................................. 2-13
VISA vs. xDM ............................................................................ 2-14

Where to Get Your Algorithms .......................................................... 2-17

CODEC Engine Details .................................................................. 2-19
RPC – Remote Procedure Call ....................................................... 2-20
Code Review ............................................................................ 2-22

Software Summary ....................................................................... 2-26

For More Information .................................................................. 2-28
Application Layer

Concepts

Goal: Accelerated Time to Market

Standard multimedia product development

Create SW foundation \rightarrow Product differentiation

The DaVinci™ Effect

Leverage DaVinci SW foundation \rightarrow Product differentiation

Shorter development cycle and/or

Leverage DaVinci SW foundation \rightarrow More time for product differentiation

Digital Multimedia System: Needs

GUI

User interface

Input (Driver) \rightarrow Master Thread

user code

Process (algorithm)

Convert input data to desired results

Output (Driver)

GP OS (eg: Linux)
- Manage multiple threads
- Assist in creation of user interface
- Manage memory
- Provide system services
Application Layer

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

Input Driver

Output Driver

GUI
Song
Volume
Bass
Treble

Process (algorithm)

- Application Host
  Software Pre-Ported to Linux
  - Device drivers
  - DSP link
  - Sample code

- Developers can
  leverage the
  abundance of available
  open source and third
  party software
  - Networking software
  - Web browser
  - Signaling software, e.g., SIP
  - Electronic
    programming guide
  - Etc.

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 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

Input Driver

Output Driver

Driver API

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");
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
// initialize IO devices...

// Execute phase
// read/swap buffer with Input device
// pass results to Output device

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

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

VISA – Four SPL Functions

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!

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 algos (discussed next)
- Author your own algos or purchase depending on your DSP needs and skills

 Linux/ARM Programmer
 create()
 control()
 VIDENC_process()
 delete()

 VISA API

 Linux/ARM Programmer
 create()
 control()
 VIDENC_process()
 delete()

 Video Imaging
 Speech Audio

 CODEC Engine
 TI authored framework

 Signal Processing Layer (SPL)
 xDM algo 1
 xDM algo 2
 xDM algo Q

 Complexity

 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 have 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);
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);
```

◆ See Chapter 9 for more details of VISA functions (i.e. prototypes)
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.
**xAIS (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).

**xAIS/xDM for Algorithm Authors**

- xDAIS provides a consistent, straightforward 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>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, dmaInit)</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>

**Instance Creation (i.e. Constructor)**

- e.g. VIDENC_create()

**Process**

- VIDENC_process()

**Delete**

- VIDENC_delete()}
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

Video
- VIDENC (encode)
- VIDDEC (decode)
- VIDANALYTICS (analysis)
- VIDTRANSCODE (transcode)

Imaging
- IMGENC (encode)
- IMGDEC (decode)

Speech
- SPHENDEC (encode)
- SPHDEC (decode)

Audio
- AUDENC (encode)
- AUDDECODEC (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
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</th>
<th>TMD320DM64x</th>
<th>TMD320DM64x</th>
<th>TMD320DM64x</th>
<th>TMD320DM64x</th>
<th>TMD320DM64x</th>
<th>OMAP35x</th>
</tr>
</thead>
<tbody>
<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>VOPF</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>VCN 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

<table>
<thead>
<tr>
<th>Audio</th>
<th>Digital Motor Control</th>
<th>Speech</th>
<th>Encryption</th>
<th>VB Modem</th>
<th>Video &amp; Imaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Stereo</td>
<td>AAC</td>
<td>Adaptive Speech Filter</td>
<td>AES</td>
<td>3-DES</td>
<td>BioKey</td>
</tr>
<tr>
<td>Decoder</td>
<td>Encoder</td>
<td>ASR Denoiser</td>
<td>Assembly DES</td>
<td>Diffe-Hellman</td>
<td>Blip Stream</td>
</tr>
<tr>
<td>Acoustic</td>
<td>Echo</td>
<td>Broadband Noise Cancel</td>
<td>ELGAMAL</td>
<td>HMAC</td>
<td>Fingerprint</td>
</tr>
<tr>
<td>Canceller</td>
<td>Adaptive Noise Canceller</td>
<td>Caller ID Text to Speech</td>
<td>MDS</td>
<td>RSA</td>
<td>Biometrics</td>
</tr>
<tr>
<td>Chorus Effect</td>
<td>MP3</td>
<td>Clear Voice Capture</td>
<td>More...</td>
<td>More...</td>
<td>H.261, H.263</td>
</tr>
<tr>
<td>MP3</td>
<td>Decoder/Encoder</td>
<td>Duplex/Noise Suppress</td>
<td>More...</td>
<td>More...</td>
<td>JPEG</td>
</tr>
<tr>
<td>More...</td>
<td>More...</td>
<td>MPEQ4</td>
<td>AES</td>
<td>More...</td>
<td>HAAR</td>
</tr>
<tr>
<td>Reverb</td>
<td>Voice Activity</td>
<td>Speech Decoder</td>
<td>RSA</td>
<td>More...</td>
<td>MJPEG</td>
</tr>
</tbody>
</table>

### 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_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

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

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

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()**

<table>
<thead>
<tr>
<th>ARM (w Linux)</th>
<th>User Space</th>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>idevfd = open(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ofilefd = open(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioctl(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CERuntime_init();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_create(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_process(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_control(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (doSvc) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(idevfd, ...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(ofilefd, ...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_delete(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(idevfd);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(ofilefd);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CODEC Engine: Engine_open()**

<table>
<thead>
<tr>
<th>ARM (w Linux)</th>
<th>User Space</th>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>idevfd = open(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ofilefd = open(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioctl(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CERuntime_init();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_create(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_process(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_control(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (doSvc) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(idevfd, ...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(ofilefd, ...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIDENC_delete(...);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(idevfd);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(ofilefd);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```c
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);
```

**VIDENC_create()**:
- Signals CE RMS to create algo instance
- CE RMS creates TSK as algo's context
- Skeleton: unpacks args from IPC

**Kernel Space**
- Input Driver
- I Buf
- Output Driver
- O Buf

**User Space**
- GUI
- DSP w/ BIOS
- MEM_alloc
- MEM_free
- xDM algo
- MEM_alloc
- algFree
- algInit
- algAlloc
- algActivate
- algDeactivate
- process

**CODEC Engine: VIDENC_control()**

```c
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);
```

**VIDENC_control()**:
- Signals myVE TSK via RPC
- myVE TSK calls DSP VIDENC_control
### CODEC Engine: VIDENC_process()

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

**CODEC engine**
- `VISA API`
- `CODEC engine`
- `Stub`
- `I Buf`
- `O Buf`
- `Kernel Space`
- `Input Driver`
- `Output Driver`

**GUI**
- `Song`
- `Volume`
- `Bass`
- `Treble`
- `mySong.mp3`

**DSP w BIOS**
- `MEM BIOS`
- `MEM_alloc`
- `MEM_free`
- `xDM algo`
- `algNumAlloc`
- `algAlloc`
- `algInit`
- `control`
- `algActivate`
- `process`
- `algDeactivate`
- `algFree`

**VIDENC_process()**
- RPC to myVE TSK: context for process()
- Drivers give buffers to algo via Shared Mem
- App is signaling center; not data buf owner

### CODEC Engine: VIDENC_delete()

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

**CODEC engine**
- `VISA API`
- `CODEC engine`
- `Stub`
- `I Buf`
- `O Buf`
- `Kernel Space`
- `Input Driver`
- `Output Driver`

**GUI**
- `Song`
- `Volume`
- `Bass`
- `Treble`
- `mySong.mp3`

**DSP w BIOS**
- `MEM BIOS`
- `MEM_alloc`
- `MEM_free`
- `xDM algo`
- `algNumAlloc`
- `algAlloc`
- `algInit`
- `control`
- `algActivate`
- `process`
- `algDeactivate`
- `algFree`

**VIDENC_delete()**
- Signals CE RMS to delete algo instance
- CE RMS also deletes algo TSK
CODEC Engine: Engine_close()

- ARM (w Linux)
  - idevfd = open(...); // create
  - ofilefd = open(...);
  - ioctl(...);
  - CERuntime_init();
  - myCE = Engine_open(...);
  - myVE = VIDENC_create(...);

- User Space
  - 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
  - Output Driver
  - I Buf
  - O Buf

- User Space
  - GUI

- DSP w BIOS
  - MEM_alloc
  - MEM_free

- CE RMS
  - VISA API
  - CODEC engine
  - Stub
  - xDM
  - algo
  - (local)

Engine_close():
- Places DSP back in reset
Software Summary

DaVinci Technology Software Architecture

Signal Processing Layer (SPL)
- Codec engine
- Video
- Imaging
- CODEC API
- I/O
- Buffers
- Shared Memory
- Available Drivers:
  - Video
  - Audio
  - File/ATA
  - USB 2.0
  - UART
  - EMAC
  - DSP/Link
  - MMC/SD
  - 2 W'dogs

User Space
- GUI
- Sound Song / incl
- Volume
- Bass
- Treble
- AV Sync
- Mux/Demux TS / ASF
- Network RTP/RTSP

Driver API
- I/O
- Buffers
- Shared Memory

Available Drivers:
- Video
- Audio
- File/ATA
- USB 2.0
- UART
- EMAC
- DSP/Link
- MMC/SD
- 2 W'dogs

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

User Space
- GUI
- Sound Song / incl
- Volume
- Bass
- Treble
- AV Sync
- Mux/Demux TS / ASF
- Network RTP/RTSP

Driver API
- I/O
- Buffers
- Shared Memory

Available Drivers:
- Video
- Audio
- File/ATA
- USB 2.0
- UART
- EMAC
- DSP/Link
- MMC/SD
- 2 W'dogs

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

Available Drivers:
- Video
- Audio
- File/ATA
- USB 2.0
- UART
- EMAC
- DSP/Link
- MMC/SD
- 2 W'dogs

User Components:
- AV Sync
- Mux/Demux TS / ASF
- Network RTP/RTSP
**APL Programmer View of DaVinci Technology System**

**ARM**
- Application Layer (APL)
- GUI
- Master thread
- Mux/Demux
- AV Sync

**User Space**
- Network RTP/RTSP
- Linux DRV API

---

**DaVinci Technology Strategy: Reduced Development Time**

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

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

*Duration (Months)*
- 32 – 50 months
- ~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
Getting Started with the Tools

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

Booting the Device ..................................................................................................................................... 3-13
   Getting to U-boot ............................................................................................................................ 3-13
   Where to find boot files .................................................................................................................. 3-14
   Das U-Boot ........................................................................................................................................ 3-16
   Configuring U-Boot ........................................................................................................................... 3-17
   Boot Variations ................................................................................................................................... 3-19
   Configure U-Boot with Tera Term Macro’s .................................................................................... 3-22

For More Information ............................................................................................................................... 3-22
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
Steps in Booting Linux

### Linux Boot Process

**Power On**

- ARM assembly code
- Passes args to Linux (bootargs)

**U-Boot**

- Initialize hardware

**Linux Kernel**

- `/sbin/init` – 1st process exe by kernel
- Login console
- Usually one of first prog's to run

**File System**

**Init Process**

**Login Prompt**

### 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 / Ubuntu: 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
  • DVSDK 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

/usr – Storage for user binaries
  • X86 Compiler for Ubuntu programs (gcc) is stored in here

Filesystems: Red Hat vs. Montavista

Red Hat (PC) MontaVista (ARM)

- Tools/Host filesystem:
  • Location for development 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:

File Management
- ls and ls -la
- cd
- cp
- ln and ln -s
- mv
- rm
- pwd
- tar (create, extract .tar and .tar.gz files)
- chmod
- chown
- mkdir
- mount, umount (in general, what is “mounting” and how do you do it?)
- alias
- touch

Program Control
- <ctrl>-c
- ps, top
- kill
- renice

Kernel
- insmod, rmmod

Network
- /sbin/ifconfig, ifup, ifdown
- ping
- nfs (What is it? How to share a folder via NFS. Mounting via NFS.)

VMware Shared Folders
- /mnt/hgfs/<shared name>

Linux Users
- root
- user
- su (... exit)

BASH
- What is BASH scripting
- What are environment variables
- How to set the PATH environment variable
- What is .bashrc? (like DOS autoexec.bat)
- man pages
- change command line prompt

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 - Linux box in a Windows PC
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

Virtual Image Used In This Workshop

Workshop VMware Image

Windows PC

Local Drive (c:\)
- Program Files
  - VMware Workstation
- vm_images
  - shared
  - TTO Workshop (ver)

Workshop VMware Images
- Screensaver & Firewall off
- NFS, TFTP, GIMP installed
- VMware toolbox installed

DM6446 Labs:
- Ubuntu 10.04
  - id = user, psw = useruser
- Software installation following DVSDK Getting Started Guide:
  - MV Linux 5.0 (Linux tools)
  - DaVinci LSP (kernel)
  - Target Filesystem

OMAP3530/AM3517 Labs:
- Ubuntu 10.04
  - id = user, psw = none
- DVSDK/SDK Tools:
  - Community Linux (Arago)
  - CodeSourcery Toolset
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 : Snapshots

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 :-)

Example of Windows ‘Guest’ Image
(Latest workshop images use Ubuntu with MV5 or community-Linux)
VMware Shared Folders

Sharing folders
- VMware shared folders
- NFS
- Samba

VMware Shared Folders
- Easiest method
- Access from: `/mnt/hgfs/shared`
Some important points in the preceding filesystem:

1. By default, all TI tools are installed into the user’s home folder – in our case, that would be the `/home/user` directory.

2. The two installers that come with DVSDK 4.x create the folders:
   - CodeSourcery – where the ARM code generation tools are located
   - `ti-dvsdk_<platform>_<dvsdk_version>` - where all the TI libraries and software are located. This directory also includes the PSP Linux directory – which contains the Linux Kernel, boot files (MLO, U-Boot), as well as the target (i.e. ARM) filesystem

3. The `psp_rebuild_omap3` folder is a replication of the Linux PSP folder. We rebuilt the Linux kernel and modules and wanted to keep our working copy in a separate folder.

4. To make things easier, we ask you to create a (Linux) symbolic link called `targetfs` to the `linux_filesys` directory (which contains the ARM root filesystem). The link will point into the `psp_rebuild_omap3` folder.

5. The `labs` and `solutions` folders will be installed by you in Lab 3. The `labs` directory contains the working files for each of the exercises in the course. As its title implies, the `solutions` directory contains the solution files.
Some important points in the preceding filesystem:

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

7. 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.
8. 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.*

9. 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.
Booting the Device

Getting to U-boot

**Booting Linux – ROM to Kernel**

1. **RBL** (ROM Internal RAM DDR2 DDR2)
2. **UBL or MLO (x-loader)**
3. **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 ulmage 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).*
Where to find boot files…

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>DM6446 EVM Default</th>
<th>AM3517 1-day Wkshp</th>
<th>Good for Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a. UBL or Xloader/MLO</td>
<td>Flash</td>
<td>MMC</td>
<td>Flash or MMC</td>
</tr>
<tr>
<td>1b. Bootloader (U-Boot)</td>
<td>Flash</td>
<td>MMC</td>
<td>Flash or MMC</td>
</tr>
<tr>
<td>2. Linux Kernel</td>
<td>Flash</td>
<td>MMC</td>
<td>TFTP (from Ubuntu)</td>
</tr>
<tr>
<td>3. Filesystem</td>
<td>Hard Drive</td>
<td>MMC</td>
<td>NFS (from Ubuntu)</td>
</tr>
</tbody>
</table>

“HDD boot”  “MMC boot”  “NFS boot”

- By default, the DM6446 DVEVM ships in “HDD boot” mode; this allows the demo applications to run “out-of-the-box”
- OMAP3530 & AM3517 ship with boot code in NAND. An MMC card demo also ships with the EVM’s. Also, the SDK provides an MMC image
- “NFS boot” (network boot) is good for application development
For MMC boot, the boot file(s) are contained on one (or two) MMC partitions. Here’s an example:

**SD / MMC Boot**

- **Partition 1** (FAT32)
  - XLDR (i.e. MLO)
  - U-Boot
  - uImage (i.e. Kernel)

- **Partition 2** (EXT3)
  - Root Filesystem

**NFS Boot**

- **Windows**
  - Tera Term

- **VMware Linux “Tools”**
  - ~/targetfs

- **Ethernet**

  - RS-232
  - U-Boot
  - EVM

  - 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:**

```
~/targetfs = /home/user/targetfs
```

Note, for the DM6446 lab exercises, we installed the tools according the the DM6446 DVEVM Getting Started Guide, which located the root filesystem at: /home/user/workdir/filesys
Das U-Boot

The Linux PSP SDK board is delivered with the open-source boot loader: Das U-Boot (U-Boot)

At runtime, U-Boot is usually loaded in on-board Flash or an SD/MMC card

In general, U-Boot performs the 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:

- Ping IP addresses
- Reads and writes arbitrary memory locations
- Uploads new binary images to memory via serial, or Ethernet
- Copies binary images from one location in memory to another
Configuring U-Boot

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
  - serverip - IP address of root file system for NFS boot
  - nfspath - Location on serverip for root filesystem
**U-Boot Macros**

At times, it can be handy to create “macros” in U-boot. These are really nothing more than variables that can be expanded when called with the U-boot “run” command.

**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
  ```

**Common U-Boot Commands and Variables**

**U-Boot Commands/Variables**

<table>
<thead>
<tr>
<th>U-Boot Commands: setenv, saveenv, printenv, run, ping, dhcp, tftp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example: setenv ipaddr 192.168.1.41</td>
</tr>
</tbody>
</table>

- **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
  - **cpclock** Available for processors with adjustable clock speeds
  - **ethaddr** Ethernet MAC address for first/only ethernet interface (eth0 in Linux); additional ethernet i/f'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

Here are a variety of ways to boot Linux, along with some of the U-Boot settings which enable them.

### Boot Variations (kernel)

<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>dhcp</td>
<td>MMC</td>
<td>MMC</td>
</tr>
</tbody>
</table>

**U-Boot's `bootcmd` variable specifies the root filesystem**

- **Flash**: `setenv bootcmd bootm 0x2050000`
- **MMC**: `setenv bootcmd mmc init; fatload mmc 0 ${loadaddr} uImage; run mmcargs; bootm ${loadaddr}`
- **TFTP**: `setenv bootcmd 'dhcp;bootm'`

### Boot Variations (filesystem)

<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>dhcp</td>
<td>MMC</td>
<td>MMC</td>
</tr>
</tbody>
</table>

**U-Boot's `bootargs` variable specifies the root filesystem**

- **HDD**: `setenv bootargs console=ttyS0,115200n8 noinitrd rw ip=dhcp root=/dev/hda1, nolock mem=120M`
- **MMC**: `setenv bootargs console=ttyS0,115200n8 noinitrd root=/dev/mmcblk0p2 rootfstype=ext3 rootwait 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 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
```
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

```
[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):$(dns2)::off
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttyS0,115200n8 noinitrd rw
   ip=$(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 DM6446)
  - tto_uboot_setup_3530.ttl

For More Information

- The Linux Filesystem Explained
Introduction

There are a wide range of tools available for ARM and ARM+DSP platforms. The list grows even longer when we include Linux and Linux distributions into our discussion of tools. Needless to say, in the short span of this chapter we’ll only be able to examine a small sampling of tools – here, we’ll try to focus on some of the tools directly supported by TI and some of its third parties.

We’ll start out by briefly discussing the types of hardware development platforms available from TI; after which we’ll examine the Software Development Kits from TI.

Next we’ll take a quick look at some of the IDE tools supporting both ARM and DSP development.

Finally, we will try to define Linux distributions, as well as provide a summary of commercial and community options supporting devices from Texas Instruments.

Along the way we’ll see that it is important to think about what your role is on your development team, as that can help us narrow down which tools will best support our development needs.

Chapter Topics

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

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

Linux Distributions ........................................................................................................................................ 4-6
  What are distributions? ............................................................................................................................... 4-6
  O/S Choices .............................................................................................................................................. 4-8
  Community Options ................................................................................................................................... 4-9
  Commercial Options .................................................................................................................................... 4-12

(Optional) How Do I Rebuild Linux ... and Why? ..................................................................................... 4-14

(Optional) Software Developer Roles & Tools ............................................................................................ 4-16
  Linux Tools ............................................................................................................................................... 4-16
  DSP Tools .................................................................................................................................................. 4-18
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
- DM365
**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 Kit**
- SDK-XAM3517-10-256512R
- Price: $199
- SW Development
  - 5" x 6"
  - Features SOM features +
    - HDMI (video only)
    - MMC/SD card slot
    - Network/USB/Serial/JTAG
    - Built-in XDS100 emulation
  - Purchase – Logic via Arrow, Avnet, Digikey
  - Support – Logic
  - SW: Linux, WinCE

**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

---

**Hardware Development Environments**

**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
## 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>TMDXEVMD3715</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>TMDXEVMD3517</td>
<td>$999</td>
<td>TI / Logic</td>
</tr>
<tr>
<td>AM18x EVM</td>
<td>TMDXEVMD1808L</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>TMDXEVMD1707</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>Beagle Board (OMAP35x)</td>
<td>Beagle</td>
<td>$149</td>
<td>Community</td>
</tr>
<tr>
<td>Hawkboard (OMAP-L138)</td>
<td>ISSPLHawk</td>
<td>$89</td>
<td>Community</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't 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>“DVSDK”</td>
<td>TI provided libraries, examples, demos Codec Engine (VISA), DSPLink, Codecs/Algos (XDM), BIOS, XDC, Linux utilities, etc.</td>
<td>All TI SOC’s: ARM, DSP, ARM+DSP</td>
</tr>
<tr>
<td>Code Gen Tools (not really “kits” per se)</td>
<td>Linux GNU Compiler (CodeSourcery)</td>
<td>All TI ARM and DSP devices where appropriate</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>
</tbody>
</table>

- 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 (PM 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

Documentation
- 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

Evaluate 802.11b/g/n and Bluetooth® capability, and begin SW development
Included in EVM box: Mar 2010
Standalone Connectivity Card upgrade available from Mistral
http://www.mistraliutions.com/WL1271
WL1271 module available from
LS Research and its distributors
http://www.mistralsolutions.com/WL1271
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**
   - CodeSourcery - GCC, GDB
   - MV DevRocket, CCSv5 (beta), 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, ARMedslack and ALIP
**O/S Choices**

- **Linux**
- **Android** (arrowboat.org)
- **WinCE**

**Build It Yourself**

**Commercial**

**Community**

**Others ...**

- QNX
- Nucleus
- BIOS
- Etc.

**Linux Distributions Options for TI**

<table>
<thead>
<tr>
<th>Custom (Build it Yourself)</th>
<th>Community</th>
<th>Commercial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom from Sources</td>
<td>TI SDK (PSP)</td>
<td>Timesys</td>
</tr>
<tr>
<td>Open Embedded (OE)</td>
<td>Ångström</td>
<td>MontaVista</td>
</tr>
<tr>
<td>“CIT” from kernel.org, and others</td>
<td>OE / GIT  Binary</td>
<td>Mentor</td>
</tr>
<tr>
<td>Bit-Bake</td>
<td>Binary Updated for each SDK release</td>
<td>RidgeRun</td>
</tr>
<tr>
<td>Recipes</td>
<td>Binary Narcissus (online tool)</td>
<td>Source</td>
</tr>
<tr>
<td></td>
<td>OE</td>
<td>Binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Update patches)</td>
</tr>
</tbody>
</table>

**Ease of Use**

- Expert User (only)
- Latest
- Tested
- Easy

**TI SDK’s for Linux, Android, WinCE is FREE**
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 – recipes 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 recipes)

Ångström : Narcissus

(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 its PSP Linux distro’s via OE

- **Build from Sources (roll your own)**
  - Build directly from sources, such as [kernel.org](http://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)

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

TI's Linux Strategy

- boot
- Linux application packages on internet
- TI Linux SDK
- TI Customers

TI Device Linux Kernel Patching Methodology

<table>
<thead>
<tr>
<th>Arago “Staging” Area</th>
<th>DaVinci and OMAP Linux staging trees</th>
<th>Mainline Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>• TI ‘work’ area</td>
<td></td>
<td>[mach-davinci]</td>
</tr>
<tr>
<td>• OMAP3x Linux patches</td>
<td></td>
<td>Official U-Boot releases</td>
</tr>
<tr>
<td>• OMAP-L1x Linux patches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• DaVinci Linux patches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• U-Boot patches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Test scripts and framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Earliest customer access to patches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accepted Patches
Commercial Options

**Commercial O/S Vendors**

- **Linux**
  - TimeSys
  - MontaVista
  - 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 stability 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 ammortize 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/DV/SDK 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
(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
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`
(Optional) Software Developer Roles & Tools

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

User Space

Driver API

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

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

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

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 to 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

Linux Development IDE Tools

<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  
                          • GNU ARM compiler (GCC)  
                          • GDB-based debug  
                          • Eclipse-based IDE | IDE debugger for driver code development  
                          KGDB based debug | • Best in class"  
                          DevRocket’s Kernel & Platform (filesystem) projects make quick and easy for this category |
| Green Hills (GHS) Multi IDE | • Also “Best in class” for Linux application development  
                          • Supports multiple OSs: INTEGRITY, Linux, etc.  
                          • GHS optimized ARM compiler | “Best in class” debug support for Linux driver development  
                          IDE debugger for driver code development  
                          Advanced debug target server – use KGDB or JTAG to connect to target | Rudimentary support for building the Linux Kernel |
| Texas Instruments Code Composer  
(CCStudio v5 Alpha) | • Alpha/beta versions are free via the TI website  
                          • Add’s GDB debugging and Linux awareness | IDE debugger for driver code development  
                          JTAG to connect to target | Not applicable |
| Open Source | • GDB  
                          • DDD  
                          • Eclipse | KGDB | Standard command-line (or ascii-based gui) support for building the linux kernel |

IDE Examples

Eclipse : CCSv5, MV DevRocket, Code Sourcery

Eclipse IDE : CCSv5, MV DevRocket, Code Sourcery
DSP Tools

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>Not Applicable</td>
</tr>
<tr>
<td>Green Hills (GHS) Multi IDE</td>
<td>Single IDE for ARM and DSP</td>
<td>TI Code Generation tools provide system configuration and building of executable DSP embedded image</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>
</tbody>
</table>

- Highly-integrated development environment
- Full DSP/BIOS aware debugging
- BIOS Visualization Tools
- Kernel Object Viewer
- BIOS Configuration Tool
- Supports ARM+DSP and DSP-only devices
- Optimizing TI Optimizing DSP Compiler
- Wizards to help with building and verifying xDAIS/xDM algorithms (i.e., codecs)

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

Code Composer Studio

- Integrated Edit / Debug GUI
- Simulator
- Code Generation Tools
- BIOS: Real-time kernel
- Real-time analysis

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,...)
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

- 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
- Debug a Linux/ARM program using GDB with CCSv5/Eclipse
Chapter Topics

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

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

Debugging with CCSv5 (Eclipse) ........................................................................................................ 5-17
Debugging & Debuggers ...................................................................................................................... 5-17
GNU Debugger ................................................................................................................................... 5-17
Graphical Debugging with GDB ........................................................................................................ 5-18
Setting Up CCSv5 for GDB ............................................................................................................... 5-19

Appendix .............................................................................................................................................. 5-21
Building Programs With gMake

Big Picture

Build Overview

Source Files “Dependencies”

Build Instructions

Executables “Targets”

app.c
app.cfg
app.h

Build Tool

app.o
app.lib

“D”
“CMD”
“T”

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.)

Looking at gcc commands...

Command Line (Examples 1-2)

“Dependencies”

“Commands”

“Targets”

app.c
app.o

Command Line

app.o
app.x86U

“D”
“CMD”
“T”

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

gcc -g -c app.c -o app.o

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

gcc -g app.o -o app.x86U

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

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:

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

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

```
gcc –g app.o –o app.x86U
```

- Becomes....

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

Creating Your First Makefile

Command Lines

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

Makefile

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

To run gMake, you can use the following commands:

- `make` (assumes the makefile name is "makefile", runs FIRST rule only)
- `make app.x86U` (specifies name of "rule" – e.g. app.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...

Looking at convenience rules...

"Convenience" Rules

- 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"):

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

Note: "all" rule is usually the first rule because if you type "make", only the first rule is executed.
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
    - gcc –g app.o –o app.x86U

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

  - `.PHONY` : clean
    - 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:

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

Examples:

```plaintext
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)
```
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 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)

```make
# -----------------------
# ------ includes -------
# -----------------------
include ./path.mak
```

User-defined variables

```make
# ------ user-defined vars -------
# --------------------------------
CC := $(X86_GCC_DIR)gcc
CFLAGS := -g
LINKER_FLAGS := -lstdc++
```

"all" rule

```make
# ------ make all -------
# -----------------------
.PHONY: all
all : app.x86U
```

Main "goal" of makefile… rule for app.x86U

```make
app.x86U : app.o
$(CC) $(CFLAGS) $(LINKER_FLAGS) $^ -o $@
@echo; echo $@ successfully created; echo
```

Intermediate .o rule. Notice the use of pattern matching.

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

Basic gMake Makefile – Review (2)

"clean" rule. Removes all files created by this makefile. Note the use of .PHONY.

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

"printvars" rule used for debug. In this case, it echos the value of variables such as "CC," "CFLAGS," etc.

```make
.PHONY : printvars
printvars:
@echo CC           = $(CC)
@echo X86_GCC_DIR  = $(X86_GCC_DIR)
@echo CFLAGS       = $(CFLAGS)
@echo LINKER_FLAGS = $(LINKER_FLAGS)
```
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

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

makefile

```make
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:

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

- -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:

```make
app.d
```

- 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.
Using Configuro to Consume Packages

Current Flow

Our current build flow looks something like this:

```
#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:

```
#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`

Let’s try using a function from a packaged library...
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)
" );
    printf ( "Hello World %d (std)\n", YYYY);
    return 0;
}
```

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

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

What does the makefile look like when using Configuro?
Using Configuro to Consume Packages

MakeFile Example – Using Configuro

```makefile
#--------------------------
#-- 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)   $<
```

How are the created files (linker.cmd, compiler.opt) used in the makefile? ...

DaVinci / OMAP Workshop - Building Programs with gMake
Looking Inside Compiler.opt & Linker.cmd

```bash
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
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
)
```
Debugging with CCSv5 (Eclipse)

Debugging & Debuggers

**Debugging & Debugger’s**

- **User Mode Debugging**
  - When debugging user mode programs, you often only want to debug – hence stop – one thread or program.
  - GDB (GNU Debugger) works well for this. (GDB discussed on next slide)
  - Connection is usually Ethernet or serial-port

- **Kernel Mode Debug**
  - Debugging kernel code requires complete system access.
  - You need KGDB (Ethernet) or scan-based (JTAG) debuggers for this.

- **A debugger lets you**
  - Pause a program
  - Examine and change variables
  - Step through code

- **Code Composer Studio (CCSv5)**
  - Latest version of TI’s graphical software debugger (i.e. IDE)
  - IDE – integrated development environment: which combines editing, building and debugging into a single tool
  - Built on Eclipse platform; can install on top of standard Eclipse
  - Allows debugging via GDB (Ethernet/serial) or JTAG (scan-based)
  - Free license for GDB debugging

GNU Debugger

**GDB : GNU Debugger**

- Open source debugger that is often supplied with the toolchain
- In this workshop, it’s included in the Code Sourcery package
- GDB has a client/server nature:
  - **GDB Server:**
    - First start `gdbserver`, specifying connection and app to be debugged
    - Server then runs your app, following gdb commands
    - Example:
      ```
      GDB Server
      Runs your app for you, based on the GDB commands you send
      ```
  - **GDB (i.e. debug client):**
    - Command-line oriented debugger
    - Telnet, Bash terminal, IDE
    - Example:
      ```
      (gdb) target remote 192.168.1.122:10000
      (gdb) step
      (gdb) b 43
      (gdb) run
      ```

DaVinci / OMAP Workshop - Building Programs with gMake
Graphical Debugging with GDB

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

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

- The target system runs:
  - Embedded OS
  - Gdbserver that controls app’s execution

Host PC

CCSv5 (Eclipse)

Target (EVM)

Ethernet

RS-232

Controls Target

Terminal

Embedded OS (ie. Linux)

GDB Server

App
Setting Up CCSv5 for GDB

1. Create/open a project and build so that you have an executable to debug

2. By default, CCSv5 doesn’t have remoteGDB debugging turned on

3. Once enabled, create a new “Debug Connection” for GDB
Setting Up CCSv5 for GDB

1. Create/open a project and build so that you have an executable to debug
2. By default, CCSv5 doesn’t have remote/GDB debugging turned on
3. Once enabled, create a new “Debug Connection” for GDB
4. Eclipse assumes “native” debugging, so you need to re-configure for remote
5. Point your connection to GDB for the target (remote) processor

6. Specify the method of connecting to the target … then click ‘Debug’
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)**

<table>
<thead>
<tr>
<th>.x rule</th>
<th>.o rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.x rule</code></td>
<td><code>.o rule</code></td>
</tr>
<tr>
<td><strong>main.x86U</strong> : <strong>main.o</strong></td>
<td><strong>main.o</strong> : <strong>main.c</strong>  <strong>main.h</strong></td>
</tr>
<tr>
<td>$(CC) $(CFLAGS) $^ $@</td>
<td>$(CC) $(CFLAGS) $&lt; $@</td>
</tr>
</tbody>
</table>

- 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
- Describe the basic file and driver I/O interfaces
- Describe what DMAI library is used for
- List the supported modules of ALSA and the DMAI functions to read and write data to it
- Build an audio pass-thru application – given audio input and output examples
Chapter Topics

Intro to Device Drivers ................................................................. 6-1

Kernel vs User Space .................................................................... 6-3

Linux Drivers - Basic Concepts ....................................................... 6-4
(1) Load the driver code into the kernel ........................................... 6-5
(2) Create a virtual file reference (node) .......................................... 6-6
(3) Mount block drivers using a filesystem ................................. 6-8
(4) Access resources using open/close/read/write ......................... 6-9

Linux Audio Drivers ..................................................................... 6-10
Open Sound System (OSS) ............................................................ 6-10
Advanced Linux Sound Architecture (ALSA) .............................. 6-11

DMAI with ALSA Driver Example ................................................ 6-12

Linux Signals .............................................................................. 6-16
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
Linux Drivers - Basic Concepts

User Access to Kernel Space

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:**

<table>
<thead>
<tr>
<th>Static (built-in)</th>
<th>Dynamic (insmod)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux Kernel</strong></td>
<td><img src="image" alt="Linux Kernel images" /></td>
</tr>
<tr>
<td>oss</td>
<td>fbdev</td>
</tr>
<tr>
<td>v4l2</td>
<td>nfsd</td>
</tr>
</tbody>
</table>

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

- 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
```

---

**Kernel Module Examples:**

- fbdev: frame buffer dev
- v4l2: video for linux 2
- nfsd: network file server dev
- dsp: oss digital sound proc.
- audio: alsa audio driver
(2) Create a virtual file reference (node)

### Linux Driver Registration

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

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

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

**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

**Block Drivers:**  
- `/dev/hda` ATA → harddrive, CF  
- `/dev/ram` external RAM

**Character Drivers:**  
- `/dev/dsp` sound driver  
- `/dev/video0` v4l2 video driver  
- `/dev/fb/0` frame buffer video driver

- **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)
- block vs. char
- Major number
- Minor number
- Name
- /dev directory
(3) Mount block drivers using a filesystem

**Mounting Block Devices**

```
# mkdir /mnt/harddrive
# ls /mnt/harddrive  "Initially empty"
# mount -t ext3 /dev/hdal /mnt/harddrive  "Now populated"
# ls /mnt/harddrive
```

- 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
(4) Access resources using open/close/read/write

Accessing Files

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

```
myFileFd = fopen("/mnt/harddrive/myfile","rw");
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 );
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 Audio Drivers

Open Sound System (OSS)

Linux OSS Driver

OSS Audio Driver (Capture example)

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

Used in older Linux distributions; e.g. MV Linux for DM6446/DM3xx

Example – Audio Capture

main.c
- setup signal handler for ctrl-c
- call audio thread
- return

audio_thread.c
- initialize OSS device (audio input setup)
- open file for capture
- create buffer to hold input
- while read input → fwrite
- cleanup

audio_input_output.c
- audio input setup
- call init mixer for line input
- call init snd dev
- init sound device
- init mixer
- audio input cleanup

Notes:
- This is the example found in Lab 6a (v2.10 labs)
- Signal handler discussed later in chapter

The signal handler in main() is covered a little later.
Advanced Linux Sound Architecture (ALSA)

**ALSA Library API**

- **Information Interface**  
  /proc/asound  
  Status and settings for ALSA driver.

- **Control Interface**  
  /dev/snd/controlCx  
  Control hardware of system (e.g. adc, dac).

- **Mixer Interface**  
  /dev/snd/mixer  
  Controls volume and routing of on systems with multiple lines.

- **PCM Interface**  
  /dev/snd/pcmCxTx  
  Manages digital audio capture and playback; most commonly used.

- **Raw MIDI Interface**  
  /dev/snd/midiCxX  
  Raw support for MIDI interfaces; user responsible for protocol/timing.

- **Sequencer Interface**  
  /dev/snd/seq  
  Higher-level interface for MIDI programming.

- **Timer Interface**  
  /dev/snd/timer  
  Timing hardware used for synchronizing sound events.

* Not implemented in current TI provided driver.

**ALSA Implementation : Block Diagram**

- **Application**  
  Sound_read()  
  Sound_write()  
  DMAI Library

- **ALSA Library**  
  arecord  
  aplay  
  amixer

- **ALSA Kernel API**  
  PCM  
  Control

- **ALSA Driver Core**  
  HW Codec Driver  
  Machine Driver  
  Platform Driver  
  McBSP Driver

- **OMAP3530**  
  Control I/F (I2C)  
  System DMA  
  McBSP

- **Audio Codec**  
  (TWL4030/AIC23)
DMAI with ALSA Driver Example

Programming Without DMAI

- Greater control / flexibility
- No new learning if already familiar with Linux and CE

Programming With DMAI

- Easier to use
- Greater portability across TI platforms
### DMAI Driver Modules

<table>
<thead>
<tr>
<th>DMAI Module</th>
<th>What it Abstracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>OSS or ALSA audio driver</td>
</tr>
<tr>
<td>Capture</td>
<td>V4L2 video capture</td>
</tr>
<tr>
<td>Display</td>
<td>V4L2 or fbdev video display driver</td>
</tr>
<tr>
<td>Resize</td>
<td>Hardware resizer driver</td>
</tr>
<tr>
<td>ColorSpace</td>
<td>Color spaces</td>
</tr>
<tr>
<td>VideoStd</td>
<td>Defines video standards</td>
</tr>
</tbody>
</table>

### DMAI Codec Engine Modules

<table>
<thead>
<tr>
<th>Adec, Aenc:</th>
<th>AUDDEC, AUDENC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idec, Ienc:</td>
<td>IMGDEC, IMGENC</td>
</tr>
<tr>
<td>Sdec, Senc:</td>
<td>SPHDEC, SPHENC</td>
</tr>
<tr>
<td>Vdec, Venc:</td>
<td>VIDDEC, VIDENC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adec1, Aenc1:</th>
<th>AUDDEC1, AUDENC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idec1, Ienc1:</td>
<td>IMGDEC1, IMGENC1</td>
</tr>
<tr>
<td>Sdec1, Sdec1:</td>
<td>SPHDEC1, SPHENC1</td>
</tr>
<tr>
<td>Vdec2, Venc1:</td>
<td>VIDDEC2, VIDENC1</td>
</tr>
</tbody>
</table>
DMAI Buffer Modules

- **Buffer** – Reference a virtually or physically contiguous memory array
- **BufferGfx** – Extends generic Buffer module with graphics attributes
- **BufTab** – Reference a table of buffers (multiple buffers)
- **Ccv** – Color space conversion of a buffer

In addition to a memory pointer, Buffer objects store the memory array’s size, bytes used, and a usage/ownership mask. BufferGfx objects additionally store x and y offset, horizontal and vertical resolution and line length.

DMAI Scheduler Modules

- **Pause**: Block a thread until an external condition occurs
- **Rendezvous**: Used to synchronize the initialization of multiple threads
Example Creation – Sound Object

Sound_Handle hSound;
Sound_Attrs sSoundAttrs = Sound_Attrs_STEREO_DEFAULT

sSoundAttrs.channels = 2;
sSoundAttrs.mode = Sound_Mode_FULLDUPLEX;
sSoundAttrs.soundInput = Sound_Input_LINE;
sSoundAttrs.sampleRate = 44100;
sSoundAttrs.soundStd = Sound_Std_ALSA;
sSoundAttrs.bufSize = 4096;

hSound = Sound_create( &sSoundAttrs );

Example Usage – Sound Object

Buffer_Attrs bAttrs = Buffer_Attrs_DEFAULT;
Buffer_Handle hBuf = NULL;

hBuf = Buffer_create( sSoundAttrs.bufSize,&bAttrs );
while( env->quit != TRUE )
{
    /* Read input buffer from ALSA input device */
    Sound_read( hSound, hBuf );

    /* Write output buffer into ALSA output device */
    Sound_write( hSound, hBuf );
}
A signal is an event generated by Linux in response to some condition, which may cause a process to take some action when it receives the signal.

- "Raise" indicates the generation of a signal
- "Catch" indicates the receipt of a signal

A signal may be raised by error conditions such as:
- Memory segment violations
- Floating-point processor errors
- Illegal instructions

A signal may be generated by a shell and terminal handlers to cause interrupts

A signal may be explicitly sent by one process to another
Signals defined in signal.h

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from abort(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
</tbody>
</table>

* Note, this is not a complete list

Raising / Catching a Signal

**Raising:**
- A foreground process can be sent the SIGINT signal by typing Ctrl-C
- Send to background process using the kill command:
  example: `kill -SIGKILL 3021`

**Receiving/Catching:**
- If a process receives a signal without first arranging to catch it, the process is terminated
- SIGKILL (9) cannot be caught, blocked, or ignored
Handling a Signal

A program can handle signals using the signal library function:

```c
#include <signal.h>

void (*signal (int sig, void(*func)(int)));
```

integer signal
to be caught or ignored

function to be called
when the specific
signal is received

---

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
        status = EXIT_FAILURE;
    }
    else
    {
        DBG("audio thread exited with SUCCESS status
    exit(status);
}

/* Callback called when SIGINT is sent to the process (Ctrl-C). */
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting..
    audio_env.quit = 1; // used as while loop condition
}
```
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
- DMAI Buffers
- DMAI Display Driver
- 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
(Optional) 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
DMAI for Video Drivers ............................................................. 7-13
DMAI Buffer Module ................................................................. 7-13
DMAI Display Module ............................................................... 7-15
Setting Video Display Properties ............................................. 7-17
DVSDK 2.xx (DM6446) ................................................................. 7-17
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

◆ 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

v4l2 Enqueue and Dequeue

- Buffers typically exist in driver’s memory space
- These buffers 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

- 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 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 Synchronization

- 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
(Optional) v4L2 Coding

**v4L2 Buffer Passing Procedure**

```c
while (cond)
    ioctl(v4l2_input_fd, VIDIOC_DQBUF, &buf);
    bufPtr = mmap(NULL,
                  buf.length,  // "start" address (usually 0)
                  PROT_READ | PROT_WRITE,  // allowed use for map'd memory
                  MAP_SHARED,  // allow sharing of mapped buffs
                  v4l2_input_fd,  // driver/file descriptor
                  buf.m.offset);  // offset requested from "start"

    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**

<table>
<thead>
<tr>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct v4l2_requestbuffers req;</td>
</tr>
<tr>
<td>req.count; // how many buffers to request</td>
</tr>
<tr>
<td>req.type; // capture, output, overlay</td>
</tr>
<tr>
<td>req.memory; // mmap, userptr, overlay</td>
</tr>
</tbody>
</table>

| 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

```
ioctl(fd, VIDIOC_REQBUFS, &req);
```

Get information on a driver buffer

```
ioctl(fd, VIDIOC_QUERYBUF, &buf);
```

Enqueue and Dequeue buffers to/from driver

```
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
- FBIOPAN_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 FBIOPAN_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
```
DMAI for Video Drivers

DMAI Buffer Module

**DMAI : Buffers and BufferGfx**

- DMAI includes a Buffer module which can create/delete buffers; a handle is returned by the `Buffer_create()` function
- Buffers objects include: length, size, attributes
- Basic Buffer attributes include: inUse, type, memory allo info
- Even further, BufferGfx (graphics buffer) extends the basic Buffer to include additional video/graphics metadata

**DMAI : BufTables**

- BufTab is an indexed set of Buffer or BufferGfx objects
- BufTab_create() can allocate an array of buffers (returning a handle)
- Check out a buffer: _get(), _getFreeBuf()
- Free buffers using: _free(), _freeAll()
Buffer Tables (BufTab)

BufTab is an indexed set of Buffer or BufferGfx objects

All buffers in a BufTab have the same size and attributes

Common BufTab Functions:

```c
hBufTab = BufTab_create( numBufs, size, *attrs )
_delete( hBufTab )
_expand( hBufTab, numBufs )
_getNumBufs( hBufTab)
```

```c
hBuf = BufTab_getBuf( hBufTab, Idx)
_getFreeBuf( hBufTab)
_freeBuf( hBuf )
_freeAll( hBufTab )
```

Simple BufTab Example

```c
#include <xdc/std.h>
#include <ti/sdo/dmai/Dmai.h>
#include <ti/sdo/dmai/BufTab.h>

BufTab_Handle hBufTab;
Buffer_Attrs bAttrs = Buffer_Attrs_DEFAULT;
Buffer_Handle hBuf;

Dmai_init(); // Always call b4 using any DMAI module

// Allocate an array of 10 basic Buffers, size 1KB
hBufTab = BufTab_create( 10, 1024, &bAttrs );

// Use the Buffers in the BufTab
hBuf = BufTab_getFreeBuf( hBufTab );
dosomething( hBuf );
BufTab_freeBuf( hBuf );

// Delete the buffer table
BufTab_delete( hBufTab );
```
DMAI Display Module

Create Display Obj Example (Driver Buffers)

```c
#define         NUM_DISPLAY_BUFS  3
Display_Attrs   sDAttrs  = Display_Attrs_O3530_VID_DEFAULT;
Display_Handle hDisplay = NULL;

::
sDAttrs.videoStd = Capture_getVideoStd(hCapture);
sDAttrs.numBufs = NUM_DISPLAY_BUFS;
sDAttrs.rotation = 90;

hDisplay = Display_create( hBufTabDisplay, &sDAttrs);
```

- Video display driver supports two types of buffers – **Driver** or **User** created

Create Display Obj Example (User Bufs)

```c
#define         NUM_DISPLAY_BUFS  3
Display_Attrs   sDAttrs  = Display_Attrs_O3530_VID_DEFAULT;
Display_Handle hDisplay = NULL;
BufferGfx_Attrs gfxAttrs = BufferGfx_Attrs_DEFAULT;
BufTab_Handle hBufTabDisplay = NULL;

::
sDAttrs.videoStd = Capture_getVideoStd(hCapture);
sDAttrs.numBufs = NUM_DISPLAY_BUFS;
sDAttrs.rotation = 90;

hBufTabDisplay = BufTab_create( NUM_DISPLAY_BUFS, bufSize,
                                BufferGfx_getBufferAttrs( &gfxAttrs ));

hDisplay = Display_create( hBufTabDisplay, &sDAttrs);
```

- Make sure the buffers are contiguous in memory – use either CMEM or DMAI's BufTab to create them.
- **User Space Buffers** are passed to the driver during open(). In our example, this is done by DMAI's Display_create() function.
Using Display Obj Example

```c
BufTab_Handle hDispBuff = NULL;

:
:
while( env->quit != TRUE )
{
    /* Acquire empty buffer from display device */
    Display_get( hDisplay, &hDispBuf );

    /* Following is not a real function, placeholder */
    myDrawDisplay( Buffer_getUserPtr( hDispBuf ),
                   Buffer_getNumBytesUsed( hDispBuf ));

    /* Pass output buffer to display device */
    Display_put( hDisplay, hDispBuf );
}
```

- Once created, using the display consists of a simple get() and put() calls
- The myDrawDisplay() function is pseudo code to just show something happening to the display buffer between the get and put

Display Driver Abstraction

- DMAI provides common if (Display_get(), Display_put()) for v4l2 and fbDev.
- FIFO module adds buffering for fbDev, so that they have a common if.
Setting Video Display Properties

DVSDK 2.xx (DM6446)

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 -nonstd 1

3. In your program (look in video_output.c and video_osd.c)
   examples:
   ```c
   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
   ```c
   fd = fopen("/sys/class/davinci_display/ch0/output",O_RDWR);
   write(fd,"COMPOSITE",10)
   ```
Page left intentionally blank.
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
Linux Processes

What is Our Main Goal?

- **Goal:** Run video and audio at the same time
- What are different ways to accomplish this?

1. **Two processors**
   - Lots of real estate and memory, costly
   - How do you sync A/V?

2. **Two processes (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

Let's compare/contrast each of these...

What is a Processor?

Processor A
Processor B
Processor C

System #1

Processor A
Memory

```
main()
  func1();
  func2();
  ...
```

Processor B
Memory

```
main()
  func3();
  func4();
  ...
```

Processor C
Memory

```
main()
  func5();
  func6();
  ...
```
**What is a Process?**

System #2

Processor A

```
Process A

main()
    func1();
    func2();
    ...  

Process B

main()
    func3();
    func4();
    ...  

Process C

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

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

Launching a Process – Terminal

```
[1] 979
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol
```
```bash
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# /lab6_soil &
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# /dev/tty/0 initialized with resolution 720x480 and 16 bpp.
```
```bash
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# ps
```
```
PIDs   TTY          TIME  CMD
975    pts/0       00:00:00 bash
979    pts/0       00:00:08 lab6_soil
```
```
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# kill 979
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# ps
```
```
PIDs   TTY          TIME  CMD
975    pts/0       00:00:00 bash
981    pts/0       00:00:00 ps
```
```
[1] 981 Terminated /lab6_soil
root@32.168.10.20:/mnt/boottcamp/lab_soil/lab6_soil/shell-Konsol# ```
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**
  - Splits one executing process into two with same environment

- **exec**
  - New process replaces old but maintains previous environment

- **fork + exec**
  - 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
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>✗</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(&new,...)`
Re-Joining Main

```
video_thread
main
```

```
new
main
```

```
pthread_create(&new,...)
```

```
pthread_exit(NULL)
```

```
main
```

```
video_thread
```

```
pthread_create(&video_thread, null, video_thread_fxn,...)
```

```
pthread_exit(NULL)
```

```
new
main
```

```
pthread_create(&new,...)
```

```
BlockWaitingForNewThreadToExit(&new)
```

```
pthread_exit(&new)
```

```
pthread_join(&new)
```

---

Multiple Threads ... With or Without Join

**Two pThreads without Join**

```
audio thread
```

```
video_thread
```

```
main()
```

```
pthread_create(&vid,...)
```

```
pthread_create(&aud,...)
```

```
pthread_exit(&vid)
```

```
pthread_exit(NULL)
```

**Two pThreads with Join**

```
audio thread
```

```
video thread
```

```
main()
```

```
pthread_create(&vid,...)
```

```
pthread_create(&aud,...)
```

```
pthread_exit(&vid)
```

```
pthread_exit(NULL)
```

```
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

devComplete = TRUE

Wasted cycles while thread A does nothing! (and thread B starves)

Thread B

Thread A
(higher priority)

Process data
Start polling for devComplete
Process data

Thread Synchronization (Polling)
Thread Synchronization (Blocking)

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

- **OSS driver**: read function is blocking
  write function blocks if outgoing buffer full
- **V4L2 driver**: VIDIOC_DQBUF ioctl is blocking
- **FBDEV driver**: FBIO_WAITFORVSYNC ioctl is blocking
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: Audio thread completes 80% of samples
Video Thread: Video thread drops 6 of 30 frames
Control Thread: 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

Audio Thread: Audio thread completes, no distortion
Video Thread: Video thread drops 1 of 30 frames
Control Thread: No user response

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

- Audio thread is guaranteed the bandwidth it needs
- Video thread takes most of remaining bandwidth
- Control thread gets a small portion of remaining bandwidth
Hybrid A/V Application Analysis

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

A good compromise:
- Audio thread completes as desired
- Video thread failure is barely perceptible
- Control thread delayed response is 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></th>
<th>SCHED_OTHER</th>
<th>SCHED_RR</th>
<th>SCHED_FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sched Method</td>
<td>Time Slicing</td>
<td>Real-Time (RT)</td>
<td></td>
</tr>
<tr>
<td>RT priority</td>
<td>0</td>
<td>1 to 99</td>
<td>1 to 99</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

```
// 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;
pthread_attr_setschedparam(&audioThreadAttrs,
    &audioThreadParams);

// Create the new thread using thread attributes
pthread_create(&audioThread, &audioThreadAttrs,
    audio_thread_fxn, (void *) &audio_env);
```
*** HTTP ERROR 404 – PAGE NOT FOUND ***
Local Codecs - Using a Given Engine

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

<table>
<thead>
<tr>
<th>Local Codecs - Using a Given Engine</th>
<th>VISA (Class) API</th>
<th>Under the Hood of the Codec Engine</th>
<th>Detailed Look at VISA Functions</th>
<th>Rules for Opening and Using Engines</th>
<th>Using Algorithms with DMAI</th>
<th>Audio</th>
<th>Video</th>
<th>Appendix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9-2</td>
<td>9-6</td>
<td>9-7</td>
<td>9-13</td>
<td>9-14</td>
<td>9-14</td>
<td>9-16</td>
<td>9-18</td>
</tr>
<tr>
<td>🎧</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fancy VIDDEC/Display Buffer Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMAI (Digital Media App Interface) Library</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codec Engine Functions Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

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);
ioctl(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

Given
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 `Engine_open()` and `_create()`**

```c
Engine_Handle myCE;
AUDENC_Handle myAE;
AUDENC_Params params;

CERuntime_init();
myCE = Engine_open("myEngine", NULL);

... setup params structure

myAE = AUDENC_create(myCE, "myEnc1", &params);
```

- Engine and Codec names are declared during the `engine config` step of the build process
- We will explore this in the next chapter

**xDM : Creation Parameters**

```c
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 secs
    XDAS_Int32 maxNoOfCh;  // max number of channels supported
    XDAS_Int32 dataEndianness;  // endianness of input data
} IAUDDEC_Params;

typedef struct IVIDDEC_Params {
    XDAS_Int32 size;  // size of this structure
    XDAS_Int32 maxHeight;  // max video height to be supported
    XDAS_Int32 maxWidth;  // max video width to be supported
    XDAS_Int32 maxFrameRate;  // max Framerate * 1000 to be supported
    XDAS_Int32 maxBitRate;  // max Bitrate to be supported in bits per second
    XDAS_Int32 dataEndianness;  // endianness of input data; def'd in XDM_DataFormat
    XDAS_Int32 forceChromaFormat;  // set to XDM_DEFAULT to avoid re-sampling
} IVIDDEC_Params;
```
Calling \_process() 

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

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 IAUDDEC\_InArgs {  
  XDAS\_Int32 size;  // size of this structure  
  XDAS\_Int32 numBytes;  // size of input data (in bytes) to be processed  
} IAUDDEC\_InArgs;  

typedef struct IAUDDEC\_OutArgs {  
  XDAS\_Int32 size;  // size of this structure  
  XDAS\_Int32 extendedError;  // Extended Error code. (see XDM\_ErrorBit)  
  XDAS\_Int32 bytesConsumed;  // Number of bytes consumed during process call  
} IAUDDEC\_OutArgs; 

Looking more closely at the BufDesc ...
Video Decoder Process Arguments

typedef struct XDM_BufDesc {
    XDAS_Int32 numBufs;
    XDAS_Int32 *bufSizes;
    XDAS_Int8 **bufs;
} XDM_BufDesc;

typedef struct IVIDDEC_InArgs {
    XDAS_Int32 size;
    XDAS_Int32 numBytes;
    XDAS_Int32 inputID;
} IVIDDEC_InArgs;

typedef struct IVIDDEC_OutArgs {
    XDAS_Int32 size;
    XDAS_Int32 extendedError;
    XDAS_Int32 bytesConsumed;
    XDAS_Int32 decodedFrameType;
    XDAS_Int32 outputID;
    IVIDEO_BufDesc displayBufs;
} IVIDDEC_OutArgs;

Buffer Descriptors

xAUDENC_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

```c
XDAS_Int8  buf0[2];
XDAS_Int8  buf1[6];
XDAS_Int8  buf2[4];
```

Allocate Buffer Descriptor

```c
XDM_BufDesc inBufDesc;
XDAS_Int32  inBufSizes[3];
XDAS_Int8   *inBufPtrs[3];
```

Build Buffer Descriptor

```c
inBufDesc.numBufs = 3;
inBufDesc.bufSizes = inBufSizes;
inBufDesc.bufs = inBufPtrs;
```

Set size & Pointer Arrays

```c
inBufSizes[0] = sizeof(buf0);
inBufSizes[1] = sizeof(buf1);
inBufSizes[2] = sizeof(buf2);
inBufPtrs[0]  = buf0;
inBufPtrs[1]  = buf1;
inBufPtrs[2]  = buf2;
```

Buffer Descriptors

VIDENC_process(myVE, &ibuf, &obuf, &myInArgs, &myRetVals);

```c
typedef struct XDM_BufDesc {
XDAS_Int32  numBufs;
XDAS_Int32  *bufSizes;
XDAS_Int8   **bufs;
} XDM_BufDesc;
```

 XDMP Buffer Descriptors are used to pass buffers to and from the _process() function

VIDENC1_process(myVE, &ibuf, &obuf, &myInArgs, &myRetVals);

```c
typedef struct XDM1_SingleBufDesc {
XDAS_Int8  *buf;
XDAS_Int32  bufSize;
XDAS_Int32  accessMask;
} XDM1_SingleBufDesc;
```

```c
typedef struct XDM1_BufDesc {
XDAS_Int32  numBufs;
XDM1_SingleBufDesc descs[];
} XDM1_BufDesc;
```

```c
Typedef enum {
XDM_ACCESSMODE_READ = 0,
XDM_ACCESSMODE_WRITE = 1
} XDM_AccessMode;
```

```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 min_inBufSize[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 min_inBufSize[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,…);
}

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](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().
Audio

Audio Encode/Decode Example (DMAI)

```c
/* Thread Execute Phase -- perform I/O and processing */
while (!env->quit)
{
    Sound_read(hSound, hBufIn);
    Aenc_process(encoderHandle, hBufIn, hBufEnc);
    Adec_process(decoderHandle, hBufEnc, hBufOut);
    Sound_write(hSound, hBufOut);
}
```

---

**Using Algorithms with DMAI**

Audio

---

Audio Encode/Decode Example (DMAI)

```
/* Thread Execute Phase -- perform I/O and processing */
while (!env->quit)
{
    Sound_read(hSound, hBufIn);
    Aenc_process(encoderHandle, hBufIn, hBufEnc);
    Adec_process(decoderHandle, hBufEnc, hBufOut);
    Sound_write(hSound, hBufOut);
}
```
Audio Encoder/Decoder Example: Variables

```c
Sound_Handle hSound = NULL;
Sound_Attrs sAttrs = Sound_Attrs_STEREO_DEFAULT;
Buffer_Handle hBufIn = NULL;
Buffer_Handle hBufOut = NULL;
Buffer_Handle hBufEnc = NULL;
Buffer_Attrs bAttrs = Buffer_Attrs_DEFAULT;
Engine_Handle engineHandle = NULL;
Aenc_Handle encoderHandle = NULL;
Adec_Handle decoderHandle = NULL;
AUDENC_Params aeParams = Aenc_Params_DEFAULT;
AUDDEC_Params adParams = Adec_Params_DEFAULT;
AUDENC_DynamicParams aeDynParams = Aenc_DynamicParams_DEFAULT;
AUDDEC_DynamicParams adDynParams = Adec_DynamicParams_DEFAULT;
```

Audio Encoder/Decoder Example: Create

```c
void *audio_thread_fxn(void *envByRef) {
    audio_thread_env *env = envByRef;
    ... 
    hBufIn = Buffer_create( blksize, &bAttrs );
    hBufOut = Buffer_create( blksize, &bAttrs );
    hBufEnc = Buffer_create( blksize, &bAttrs );
    engineHandle = Engine_open( env->engineName, NULL, NULL );
    encoderHandle = Aenc_create( engineHandle, "encoder_name",
                                &aeParams, &aeDynParams );
    decoderHandle = Adec_create( engineHandle, "decoder_name",
                              &adParams, &adDynParams );
}
```

Audio Encoder/Decoder Example: Process

```c
/* Thread Execute Phase -- I/O and processing */
while ( !env->quit ) {
    Sound_read( hSound, hBufIn );
    Aenc_process( encoderHandle, hBufIn, hBufEnc );
    Adec_process( decoderHandle, hBufEnc, hBufOut );
    Sound_write( hSound, hBufOut );
}
```
/* Thread Execute Phase -- I/O and processing */
while ( !env->quit )
{
    Capture_get( hCap, &cBuf );
    Venc_process( hEnc, cBuf, encBuf );
    Capture_put( hCap, cBuf );
    Display_get( hDisp, &dBuf );
    Vdec_process( hDec, encBuf, dBuf );
    Display_put( hDisp, dBuf );
}
Video Encoder Object – Create Example

```c
Engine_Handle hEng = NULL;
Venc_Handle hEnc = NULL;
VIDENC_Params sMyEncParams = Venc_Params_DEFAULT;
VIDENC_DynamicParams sMyEncDynParams = Venc_DynamicParams_DEFAULT;

void *video_thread_fxn(void *envByRef) {
    audio_thread_env *env = envByRef;
    ...
    hEng = Engine_open( env->engineName, NULL, NULL );
    hEnc = Venc_create( engineHandle ,
                        "encoder_name",
                        &sMyEncParams ,
                        &sMyEncDynParams );
}
```

Video Decoder Object – Create Example

```c
Engine_Handle engineHandle = NULL;
Vdec_Handle decoderHandle = NULL;
VIDDEC_Params sMyDecParams = Vdec_Params_DEFAULT;
VIDDEC_DynamicParams sMyDecDynParams = Vdec_DynamicParams_DEFAULT;

engineHandle = Engine_open( env->engineName, NULL, NULL );
decoderHandle = Vdec_create( engineHandle ,
                             "decoder name",
                             &sMyDecParams ,
                             &sMyDecDynParams );
hBufTabDecoder = BufTab_create( NUM_DECODER_BUFS, bufSize,
                                BufferGfx_getBufferAttrs(&gfxAttrs) );
Vdec_setBufTab( decoderHandle, hBufTabDecoder );
```

Audio Encode/Decode Example (DMAI)

```
Capture_get(h, &cBuf)
Capture_put(h, cBuf)
Venc_process(h, cBuf, encBuf)
Display_get(h, &dBuf)
Display_put(h, dBuf)
```
Fancy VIDDEC/Display Buffer Management

How Does v4l2 Reference Buffers

```c
typedef struct v4l2_buffer{
    _u32 index;
    enum v4l2_buf_type type;
    _u32 bytesused;
    _u32 flags;
    _u32 sequence;
    enum v4l2_field field;
    struct timeval timestamp;
    struct v4l2_timecode timecode;
} v4l2_buffer;
```

- V4L2 driver uses/knows a buffer's physical address
- To determine the application's virtual address requires a cycle-intensive MMU check
- Instead, V4L2 driver queues and dequeues buffers by `index`

Video Decoder Indexing

- Encoded (compressed) video data may contain bi-directional "B" frames which reference data from previous and following video frames
- This means that video decoders may end up decoding video frames out-of-order
Video Decoder Indexing Details

// pseudo code
Vdec_process(Buffer1);
Vdec_process(Buffer2);
Vdec_process(Buffer3);

BufTab Sharing Example - Initialization

/* Open codec engine and create a video decoder instance */
engineHandle  = Engine_open( env->engineName, NULL, NULL );
decoderHandle = Vdec_create( engineHandle, VIDEO_DECODER,
    &dParams, &dDynParams );

/* Create a BufTab to use with decoder and display driver */
hBufTabDecoderDisplay = BufTab_create( NUM_DECODER_DISPLAY_BUFS,
    bufSize,
    BufferGfx_getBufferAttrs(&gfxAttrs));

/* Set hBufTabDecoderDisplay for use with video decoder */
Vdec_setBufTab( decoderHandle, hBufTabDecoderDisplay );

/* Initialize Display driver with hBufTabDecoderDisplay */
hDisplay = Display_create( hBufTabDecoderDisplay, &dAttrs );
BufTab Sharing Example – Buffer Passing

```c
while ( !env->quit ) {
    fread( Buffer_getUserPtr(inBuf), 1 ,
           Buffer_getSize(inBuf),   hFile );
    Display_get( hDisplay, &dBuf );
    Vdec_process( decoderHandle, inBuf, dBuf );

    dBuf = Vdec_getDisplayBuf( decoderHandle );

    while( dBuf != NULL ) {
        Display_put( hDisplay, dBuf );
        dBuf = Vdec_getDisplayBuf( decoderHandle );
    }
}
```

- `dBuf` (BufferGfx_Handle) can be passed between Display and Vdec because both were initialized with `hBufTabDecoderDisplay`
- `Vdec_process()` and `Vdec_getDisplayBuf()` act as “put” and “get” (respectively) for video decoder queue
- If decoder “get” doesn’t return a buffer to display, the display driver queue is not updated on this pass of the loop
- Conversely, if decoder “get” provides a non-NULL handle for `dBuf`, then we’ll keep que’ing buffers from the decoder to the display until none are left

DMAI (Digital Media App Interface) Library

This shows mapping between our DM6446 lab exercises and DMAI-based labs.

Workshop Example – Lab06b with Audio Decode

- `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, "aacdec", &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;
}
```

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](http://gforge.ti.com/gf/project/dmai)
### Codec Engine Functions Summary

#### 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.</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_getLastError()</td>
<td>Get the error code of the last failed operation.</td>
</tr>
<tr>
<td>Engine_getUsedMem()</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>

#### 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

(Optional) Building an Engine Deliverable ......................................... 10-9
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.**

![](image-url)

- **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.evdm6446
  - Target – e.g. gnu.targets.MVArm9
  - .cfg – indicates which packages to include → Which packages should we include?
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", [
    {name: "video", mod: vidDec, local: true},
    {name: "audio", mod: audDec, local: true},
]);
```
Local Only Engine for ARM

Engine Configuration File (.cfg file)

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},
]);

Example: 'tto.codecs.auddec.AUDDEC'
Package Name: tto.codecs.auddec
Codec Module Name: AUDDEC
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

Engine configuration file (.cfg)

```javascript
var myEng = Engine.create("myEngine", [
  {name: "video", mod: vidDec, local: true},
  {name: "audio", mod: audDec, local: true},
]);
```

Application Source File (app.c)

```javascript
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);
```
(Optional) Building an Engine Deliverable

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

/* ======== 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

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.
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......................................................................................................................................11-2

Server Management Layer (VISA functions)...........................................................................................11-6

Priorities on the DSP Server.....................................................................................................................11-7

Modifying the Engine to use a Remote Server .......................................................................................11-9

Append Algo’s to an Engine created From Server .................................................................................11-11

Contiguous Memory Functions.................................................................................................................11-12
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 DSP

Stack

App locals

c
b
a

Algo code

Remote Procedure Call “RPC”

Master Thread / App code

```
... VIDENC_process(a, b, c);
...
```

```
process(a, b, c) {
...
...
}
```

ARM DSP

Stack

locals

In addition to packaging variables for transport to the DSP, the stub is responsible for any necessary address translation of ptrs
Remote Codec Calls

Remote Procedure Call “RPC”

- In addition to packaging variables for transport to the DSP, the stub is responsible for any necessary address translation of ptrs.
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

**DSP Server : Priority Based Scheduling**

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 vidDec = xdc.useModule('codecs.viddec.VIDDEC');
var vidEnc = xdc.useModule('codecs.videnc.VIDENC');

var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
    {name: "myEnc1", mod: vidDec, local: false},
    {name: "myEnc2", mod: vidEnc, local: false},
]);

myEng.server = "./myServer.x64P";
```

**New Method is:**
- Shorter (i.e. easier)
- Less error prone
- Configuro can extract additional info from package metadata (e.g. memory map)

**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 file, inside the server's package
    "tto.servers.example" ); // Server's package name
```
If you are not planning to put the ARM (.xv5T) and DSP (.x64P) executables into the same directory in the ARM/Linux filesystem, then you should set the .server property of your engine. You can see this in the following slide, though we won’t need to do this in our upcoming lab exercise – though it doesn’t hurt to add it to your configuration file.

**Create Engine from Server**

<table>
<thead>
<tr>
<th>Old Method</th>
<th>New Method is:</th>
</tr>
</thead>
</table>
| var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.DSPLINK_LINUX;
var vidDec = xdc.useModule('codecs.vddec.VIDDEC');
var vidEnc = xdc.useModule('codecs.vdenc.VIDENC');
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEng = Engine.create("myEngine", [
  {name: "myEnc1", mod: vidDec, local: false},
  {name: "myEnc2", mod: vidEnc, local: false}]);
myEng.server = "./myServer.x64P"; | ◆ Shorter (i.e. easier)
◆ Less error prone
◆ Configuro can extract additional info from package metadata (e.g. memory map) |

<table>
<thead>
<tr>
<th>New Method</th>
<th></th>
</tr>
</thead>
</table>
| 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", ...
... "./myServer.x64P", ...
... "tto.servers.example" );
myEng.server = "./myServer.x64P"; | myEng.server is not needed in our upcoming lab, since we’ll put both executables (.xv5T, .x64P) into the /opt/workshop directory.

// Loc'n of server exe at runtime, relative to ARM program;
// only needed if not found in the same folder as ARM.
Append Algo’s to an Engine created From Server

Append Local Algo’s to Engine.createFromServer()

```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 the .x64P file is, inside the server's pkg
    "tto.servers.example" ); // Server's package name

var audDec = xdc.useModule('codecs.auddec.AUDDEC');
var audEnc = xdc.useModule('codecs.audenc.AUDENC');

```

Add local algo’s, even when using .createFromServer()

- Reference the codec's package (i.e. xdc.useModule)
- Append each codec to the Engine's array of algorithms
- Remember to set “local” to “true”

See the following wiki page for more details:

http://processors.wiki.ti.com/index.php/Configuring_Codec_Engine_in_Arm_apps_with_createFromServer
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>ptr 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 mgmt unit). Since most DSP's 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:

**Standard C syntax**

```c
#include <stdlib.h>
define SIZE 128
a = malloc(SIZE);
a = {...};
use buffer...
free(a);
```

**Using Memory_contig() functions**

```c
#include <ti/sdo/ce/osal/Memory.h>
define SIZE 128
a = Memory_contigAlloc(SIZE, ALIGN);
a = {...};
use buffer...
Memory_contigFree(a, SIZE);
```

In our upcoming lab exercise we’ll need to change to contiguous memory allocations in our ARM/Linux C code so that the buffers we pass to the DSP will be continuous physically, which the DSP requires.

Also, since the buffers will come from the CMEM region of the memory map, they’ll also be accessible to ARM+DSP devices that wrap a MMU around the DSP, like the OMAP3530.

We could have easily used the Memory_contigAlloc() functions for our earlier lab exercises, but we wanted to highlight the change in this chapter/lab, since this is where it becomes required.
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

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
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 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

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 ...

Codecs
RMS
Instance
Engine SPI
Skeleton
VISA API
xDM Codec
xDM Codec

DSP Server

DSP Link

DSP BIOS
What Makes Up a DSP Server

Building a Server

Source Files

- main.c
- BIOS Config (.tcf)
- packages

Build Instructions

Executable File

- XDC
- Package .x64P

$> XDC

✓ 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");

    /* Print it! */
    GT_0trace(gtMask, GT_4CLASS, "main> Welcome to DSP server's main()");
}
```

◆ 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

- main.c
- BIOS Config (.tcf)

Build Instructions

Executable File

- Package
  - x64

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)

```plaintext
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/initializes 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
What Makes Up a DSP Server

Building a Server

Source Files
- packages
- main.c
- BIOS Config (.tcf)

Build Instructions

Executable File
- Package .x64P

$> 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?

✓ 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

Source Files
- packages
- main.c
- BIOS Config (.tcf)

Build Instructions

Executable File
- Package .x64P

$> XDC

- package.xdc - name, version, dependencies
- package.xs - build time interfaces
- package.bld - build instr's (gcc, options, profiles, etc.)
- server.cfg - specifies input pkg's
DSP Server Wizard

Build DSP Server Using Codec Engine Wizard

- Choose Platform:
- 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

Clicking Next...

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:

What does the wizard create?
### 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

Let's look at the server.cfg file...

server.cfg

Setup OSAL (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. */
Server.threadAttrs.stackSize = 16384;
Server.threadAttrs.priority = Server.MINPRI;

/* Import the algorithm's configuration */
utils.importFile("codec.cfg");```

Let's look at the codec.cfg file...

codec.cfg
Building a Server – Config (.cfg) Files

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");
```

Looking more closely at 'codec.cfg'...

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", // C name for the of codec
        mod: V_COPY, // Javascript var reference for module
        threadAttrs: {
            stackSize: 1024, // Size of task’s stack
            stackMemId: 0, // BIOS MEM segment ID for task’s stack
            priority: Server.MINPRI + 1 // Task’s priority
        },
        groupid: 1 // Scratch pool (includes: mem, dma, etc.)
    },
];
```

- 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.)

Examining the "groupid"...
Memory – Scratch Groups

Groups of Scratch Memory

- Low Priority Algos (Priority = 0)
  - Algo A1
  - Algo A2
  - GroupID 0

- 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

Blockwise Representation
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.

- **General Setup**
  - Add OSAL module
  - Add server module
  - Include codec.cfg

- **Memory Setup (DSKT2 module)**
  - Associate IALG memory types with available target memory
  - Specify default scratch group memory sizes

- **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

### 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"
};```

How was memory defined? (server.tcf)
DM6446 EVM Memory Map

- **L1DSRAM**
  - Level 1 On-Chip SRAM (0 waitstate)
  - 16K cache
  - 64K sram
  - (All 64K of L2 is config’d as cache)

- **Linux**
  - 120M
  - Set via Linux command line to MEM=120M

- **CMEM**
  - 8M
  - Set with CMEM insmod command

- **DSP Heap (DDRNLGHEAP)**
  - 122M
  - Set in BIOS Textual Config (.tcf) file

- **App Prog (DDR)**
  - 4M
  - Set in BIOS Textual Config (.tcf) file

- **DSP Link**
  - 1M
  - Set in BIOS Textual Config (.tcf) file

- **Reset/Int**
  - 1M
  - Set in BIOS Textual Config (.tcf) file

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 the Memory Map...
Connecting Spaces to Memory (i.e. what to give DSKT2)

“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

How do I config DSKT2?

L1DSRAM
- Level 1 On-Chip SRAM (0 waitstate)
  - 16K cache
  - 64K sram
  (All 64K of L2 is config’d as cache)

Linux
- 120M

CMEM
- 128M

DDRALGHEAP
- 250M

App Prog
- 254M

DSP Link
- 255M

Reset
- 256M

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
DSKT2 Module

Initialized during Configuration

SRAM:
0x8000_0000-
0x8010_0000

IRAM:
0x0000_0000-
0x0004_0000

Usage Phase (run-time)

DSKT2

IRAM

SRAM

Alg1:
20K SRAM,
5K IRAM

Alg2:
10K SRAM,
10K IRAM

- 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

DSKT2 Setup in server.cfg

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?

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Req.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algo B1</td>
<td>4K</td>
<td>4K</td>
</tr>
<tr>
<td>Algo B2</td>
<td>2K</td>
<td></td>
</tr>
<tr>
<td>Algo B3</td>
<td>3K</td>
<td>4K</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>=</td>
<td>4K</td>
</tr>
</tbody>
</table>

- 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 ...

In this case, largest alloc came last. Thus, when requested, 4K block won't fit
A new 4K alloc must be done

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   = "DDRLGHEAP";
DSKT2.IPROG    = "L1DSRAM";
DSKT2.EPROG    = "DDRLGHEAP";
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/ALG 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.
C6EZ Tools

DSP Development made easier with C6EZ Tools

Seamlessly ports ARM code to DSP
(ARM Developers)

Provides ARM access to ready-to-use DSP kernels
(System Developers)

Graphical Software Development Tool for quick prototyping
(DSP Developers)

What can a System Developer do on the DSP?

- Offload key signal-processing and math intensive algorithms
- Leverage some DSP features, like floating-point computations
- Free up ARM MIPS for additional system features
- Save money by not moving to a more powerful & expensive ARM!

4 Ways to access the DSP from the ARM

1. Call DSP directly over DSPLINK/SYLINK
2. Wrap the algorithm with IUniversal interface to Codec Engine
3. Use C6EZRun to cross compile and redirect code for DSP
4. Use C6EZAccel to call DSP realtime-kernels/functions from the ARM
**C6Run**

**C6EZRun for ARM® Developers**

- **C6EZRun** quickly ports ARM code to run on the DSP, making DSP development easy, fast and cost efficient

  - Compile C code for DSP execution right from the Linux shell
  - No modification to ARM application code required
  - Provides a familiar development environment with a GCC-like interface, simplifying the user/development experience

---

**C6Accel**

**C6EZAccel for System Developers**

- **C6EZAccel** provides ARM application access to production ready, performance tuned functions on the DSP

  - ARM side API with easy access to 100's of optimized DSP kernels
  - Asynchronous execution allows ARM/DSP parallel processing
  - Includes key signal processing, image processing and math functions
C6EZAccel application

System On Chip

- DSP server collating DSPLink, CMEM, DSP/BIOS, and DSP libraries
- ARM user interface is a static library that abstracts the codec engine setting and ARM side cache management
- Function is retargeted to DSP, with error handling managed on ARM/Linux using error codes

C6EzAccel Processing Blocks

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Now</th>
<th>2Q11</th>
<th>4Q11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation Signal Processing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital Signal Processing</td>
<td>32</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>Image Processing</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>Floating Point Math</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Fixed Point Math</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Filter Package</td>
<td></td>
<td>128</td>
<td></td>
</tr>
<tr>
<td><strong>Application Specific Software</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vision And Analytics Library (VLIB)</td>
<td>52</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Open Source Computer Vision (OpenCV)</td>
<td>60</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>ProAudio: Audio Processing Library</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Power and Energy</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td><strong>Total Supported Functions</strong></td>
<td>134</td>
<td>269</td>
<td>497</td>
</tr>
</tbody>
</table>
C6EZAccel DSP server

- C6EZAccel algorithm is a collection of DSP optimized libraries combined into a single xDAIS algorithm
- Implements iUniversal interface to the Codec Engine
- Server integrates BIOS, Link/Code-Engine, codecs and C6Accel into DSP executable
- C6Accel includes Unit Servers provided in the package
- Codec Servers with 6EZAccel integrated as of SDK 4.x

C6EZAccel Application Usage

**Headers and definitions**

```c
C6accel_Handle hC6accel = NULL
#define ENGINENAME "<platform_name>"
#define ALGNAME "c6accel"
#include "soc/c6accelw/c6accelw"
```

**Source code to call fxn1 and fxn2 on DSP**

```c
CE_Runtime_init();
hC6accel = C6accel_create();
............... Status = C6accel_fxn1(hC6accel,parameters1);
............... Status = C6accel_fxn2(hC6accel,parameters2);
............... C6accel_delete(hC6accel);
```

- Creates a C6accel handle.
- Calls fxn1 and fxn2 on the DSP.
- Deletes C6Accel handle.

**User Experience in ARM Application**

1. Analyze source to determine what to move to the DSP
2. Allocate all buffers to be passed to DSP using CMEM
3. Offload function to the DSP by invoking C6EZAccel APIs or CE APIs
4. Integrate DSP codec server with application using XDC configuration
5. Compile all ARM code with the linker command file generated from Step 4
C6EAccel: Key Features

- **Asynchronous Calling**
  - Parallel processing on ARM and DSP
  - Frees ARM MIPS, allows feature addition
  - Maximum system level performance
  - Source to call fxn1 asynchronously on DSP

- **Chaining of APIs**
  - Averaging of inter processor overhead
  - Improved Efficiency to make multiple calls

---

Performance benefits of offloading tasks to the DSP

**C6EZAccel vs Native ARM:**

- *FFT function*

**C6EZAccel Overheads:**

- Cache invalidation on ARM and the DSP
- Address translation from virtual to/from physical
- DSP/SYSLINK overhead.
- Activating, processing, deactivating the C6EZAccel algorithm on the DSP

**DSP provides higher gains on larger images/data**
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 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

```javascript
var Pkg = xdc.useModule('xdc.bld.PackageContents');
/*
 * ======== Add Executable ========
 */
print( "Print something to output while building package." );
Pkg.otherFiles = [ "readme.txt", "document.pdf" ];
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

```javascript
Parts of config.bld:
- DSP Target
- ARM Target
- Linux Host Target
- Build Targets

- `config.bld` defines the build targets for your projects
- Build targets settings include: tool locations, compiler settings, etc.
- When, in `package.bld`, you specify to build for a particular target, XDC refers to `config.bld` to resolve this info
```

- `config.bld` defines the build targets for your projects
- Build targets settings include: tool locations, compiler settings, etc.
- When, in `package.bld`, you specify to build for a particular target, XDC refers to `config.bld` to resolve this info

```javascript
// ======== Linux host target ========
var Linux86 = xdc.useModule('gnu.targets.Linux86');
Linux86.rootDir = "/usr";

// ======== Arm target ========
var MVArm9 = xdc.useModule('gnu.targets.MVArm9');
MVArm9.rootDir = "/devkit/arm/v5t_le/bin/arm_v5t_le-";

// ======== DSP target ========
var C64P = xdc.useModule('ti.targets.C64P');
C64P.platform = "ti.platforms.emvDM6446";
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];
```

- `config.bld` defines the build targets for your projects
- Build targets settings include: tool locations, compiler settings, etc.
- When, in `package.bld`, you specify to build for a particular target, XDC refers to `config.bld` to resolve this info

```javascript
// ======== Linux host target ========
var Linux86 = xdc.useModule('gnu.targets.Linux86');
Linux86.rootDir = "/usr";

// ======== Arm target ========
var MVArm9 = xdc.useModule('gnu.targets.MVArm9');
MVArm9.rootDir = "/devkit/arm/v5t_le/bin/arm_v5t_le-";

// ======== DSP target ========
var C64P = xdc.useModule('ti.targets.C64P');
C64P.platform = "ti.platforms.emvDM6446";
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];
```
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

Outline

- Introduction
  - What is xDAIS (and VISA)?
  - Software Objects
  - Master Thread Example
  - Intro to Codec Engine Framework (i.e. VISA)
- Algorithm Lifecycle
- Frameworks
- Algorithm Classes
- Making An Algorithm
- Appendix
**Chapter Topics**

**Authoring a xDAIS/xDM Algorithm** ........................................................................................................... 13-1

*Introduction* .................................................................................................................................................. 13-3
  - What is xDAIS (and VISA)? ............................................................................................................. 13-3
  - Software Objects .......................................................................................................................... 13-4
  - Master Thread Example.................................................................................................................. 13-6
  - Intro to Codec Engine Framework (i.e. VISA).................................................................................. 13-6

**Algorithm Lifecycle** ...................................................................................................................................... 13-9
  - Create............................................................................................................................................... 13-10
  - Process............................................................................................................................................. 13-14
  - Delete............................................................................................................................................... 13-16

**Frameworks** .................................................................................................................................................. 13-17
  - Algorithm Classes ........................................................................................................................... 13-19
  - xDM/VISA Classes .......................................................................................................................... 13-19
  - Universal Class............................................................................................................................... 13-20
  - Extending xDM ............................................................................................................................... 13-21
  - Design Your Own – Extending the Universal Class........................................................................ 13-23

**Making an Algorithm** ..................................................................................................................................... 13-24
  - Rules of xDAIS ............................................................................................................................... 13-24
  - Using the Algorithm Wizard ........................................................................................................... 13-26

**Appendix** ...................................................................................................................................................... 13-30
  - Reference Info ............................................................................................................................... 13-30
  - Using the Old Algorithm Wizard .................................................................................................... 13-31
  - (Optional) xDAIS Data Structures................................................................................................. 13-39
  - (Optional) Multi-Instance Ability .................................................................................................... 13-43
  - (Optional) xDAIS : Static vs Dynamic ........................................................................................... 13-44
Introduction

What is xDAIS (and VISA)?

Algo’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

Software Components/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

```c
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
```

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)**

- **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 if) **New!**

  - `V` VIDEENC
  - `I` IMGENC
  - `S` SPFENC
  - `A` AUDENC
  - `O` VIDANALYTICS
  - `U` VIDTRANSCODE
  - `V` 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 if) **New!**

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

---

**Complexity**

Reducing dozens of functions to 4
Master Thread Key Activities

```c
// Create Phase
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, &dyParams, &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

```javascript
/* 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**

Codec Engine (Application Framework)

1. Here’s the way I want you to perform…
   ```c
   Params = malloc(x);
   *Params= PARAMS;
   ```

---

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:

  ```
  typedef struct IFIR_Params {
    Int size;          // size of params
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
  } IFIR_Params;
  ```

Instance Creation - start

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)

   ```
   MemTab = malloc(5*N)
   ```

4. I’ll make a place where you can tell me about your memory needs…
**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:

  ![Memory Table Diagram]

  **MemTab Example:**

  - Based on the four memory details in MemTab,
  - Application allocates each memory block, and then
  - Provides base address to MemTab

  ![Algorithm Diagram]

  - Algo provides info for each block of memory it needs,
  - Except base address …
### Instance Creation - finish

Algorithm

5. Tell me about your memory requirements...

6. I’ll enter my needs for each of the N blocks of memory, given these parameters, into the MemTab...

7. I’ll go get/assign the memory you need...

8. Prepare the new instance to run!

MemTab

for (i=0; i<=N; i++)

mem = malloc(size);

9. Initialize vars in my instance object using Params & Base’s...

10. Delete MemTab

Example: Params & InstObj

#### 1. Creation Params

typedef struct IFIR_Params {
    XDAS_Uint32 size;
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
} IFIR_Params;

#### 2. memTab

Size
Alignment
Space
Attributes
IFIR_Obj Address

Size
Alignment
Space
Attributes
Block Address

Size
Alignment
Space
Attributes
History Address

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;
Process

Instance Execution

Codec Engine
(Application Framework)

1. Get ready to run.
Scratch memory is yours now.

Algorithm

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>A</td>
<td>Scratch A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Scratch B</td>
<td></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>A</td>
<td>Scratch A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Scratch B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Scratch C</td>
<td></td>
<td></td>
<td></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

**1K Block**
(data to process)

**32 tap FIR**

**Instance Execution**

**Codec Engine**
(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
Delete

Instance Deletion

If I no longer need the algorithm:
1. I’ll make a memTab again, or reuse the prior one
   \[ \text{*memTab} = \text{malloc}(5*N) \]
2. What memory resources were you assigned?
3. Re-fill memTab using \text{algAlloc} and base addresses stored in the instance object
4. \text{free} all persistent memories recovered from algorithm

Codec Engine
(Application Framework)

Algorithm

InstObj
Param1
Param2
...
Base1
Base2
...

MemTab
size
alignment
space
attrs
*base

algFree()
Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>algNumAlloc, algAlloc, algInit</td>
</tr>
<tr>
<td>Process</td>
<td>process</td>
</tr>
<tr>
<td>Delete</td>
<td>algFree</td>
</tr>
</tbody>
</table>

- If all algorithms must use these ‘create’ functions, couldn’t we simplify our application code?

Framework Create Function

<table>
<thead>
<tr>
<th>Create Functions</th>
<th>Codec Engine Framework (VISA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>algNumAlloc ()</td>
<td>VIDENC_create()</td>
</tr>
<tr>
<td>algAlloc ()</td>
<td></td>
</tr>
<tr>
<td>algInit ()</td>
<td></td>
</tr>
</tbody>
</table>

- These functions are common for all xDAIS/xDM algo’s
- **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) algos
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
- 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 Algorithm: DSP Algo Author**

- algNumAlloc
- algAlloc
- alginit
- 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!

xDAIS/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

```c
typedef struct IUNIVERSAL_Params {
    XDAS_Int32 size;
} IUNIVERSAL_Params;
```

```c
//====================================================================

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

XDAS_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 IAACDEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
    Int DownSampleSbr; // AAC specific
} IAACDEC_DynamicParams;

typedef struct IAACDEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
} IAACDEC_InArgs;

typedef struct IAACDEC_OutArgs{
    IAUDDEC_OutArgs auddec_outArgs;
} IAACDEC_OutArgs;

typedef struct IMP3DEC_Params {
    IAUDDEC_Params auddec_params;
} IMP3DEC_Params;

typedef struct IMP3DEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
} IMP3DEC_DynamicParams;

typedef struct IMP3DEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
    XDAS_Int32 offset;
} IMP3DEC_InArgs;

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
    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 *ouBuFs,
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

genCodecPkg Wizard (1)

genCodecPkg Wizard (2)
Files Created by genCodecPkg

Code Created: Functions

```c
/*
 * ======== MIXER_TTO_IMIXER ========
 * This structure defines TTO's implementation of the IUNIVERSAL
 * interface for the MIXER_TTO module.
 */
IUNIVERSAL_Fxns MIXER_TTO_IMIXER = {
    {IALGFXNS},
    MIXER_TTO_process,
    MIXER_TTO_control,
};
```
Code Created: algAlloc()

```c
Int MIXER_TTO_alloc( const IALG_Params *algParams,
                     IALG_Fxns **pf, IALG_MemRec memTab[])
{
    /* Request memory for my object */
    memTab[0].size      = sizeof(MIXER_TTO_Obj);
    memTab[0].alignment = 0;
    memTab[0].space     = IALG_EXTERNAL;
    memTab[0].attrs     = IALG_PERSIST;

    return (1);
}
```

Code Created: algInit

```c
Int MIXER_TTO_initObj( IALG_Handle handle, const IALG_MemRec memTab[],
                       IALG_Handle parent, const IALG_Params *algParams)
{
    const IMIXER_Params *params = (IMIXER_Params *)algParams;
    /*
    * Typically, your algorithm will store instance-specific details
    * in the object handle. If you want to do this, uncomment the
    * following line and the 'obj' var will point at your instance object.
    */
    //    MIXER_TTO_Obj *obj = (MIXER_TTO_Obj *)handle;

    /*
    * If no creation params were provided, use our algorithm-specific ones.
    * Note that these default values _should_ be documented in your algorithm
    * documentation so users know what to expect.
    */
    if ( params == NULL ) {
        params = &IMIXER_PARAMS;
    }

    /* Store any instance-specific details here, using the 'obj' var above */
    return (IALG_EOK);
}
```
/* Local casted variables to ease operating on our extended */
IMIXER_InArgs *inArgs   = (IMIXER_InArgs *)universalInArgs;
IMIXER_OutArgs *outArgs = (IMIXER_OutArgs *)universalOutArgs;

/* Note that the rest of this function will be algorithm-specific */
/* the initial generated implementation, this process() function
* copies the first inBuf to the first outBuf. But you should
* this to suit your algorithm's needs. */

/* report how we accessed the input buffers */
/* report how we accessed the output buffer */
return (IUNIVERSAL_EOK);
Appendix

Reference Info

References

- Codec Engine Algorithm Creator User’s Guide
  SPRUED6  Texas Instruments

- Codec Engine Server Integrator’s Guide
  SPRUED5  Texas Instruments

- xdctools_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>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<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>
Using the Old 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
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 secs
    XDAS_Int32 maxNoOfCh; // max number of channels supported
    XDAS_Int32 dataEndianness; // endianness of input data
} IAUDDEC_Params;

Void (*algActivate) (IALG_Handle);
Int (*algAlloc) (const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
Void (*algDeactivate) (IALG_Handle)
Int (*algFree) (IALG_Handle, IALG_MemRec *)
Int (*algInit) (IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
Void (*algMoved) (IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *)
Int (*algNumAlloc) (Void)

TypeDef struct IOMAP_Params {
    Int size;
    Int XOMAP_ImgProc;
    Int XOMAP_ImgProcEx;
    Int XOMAP_Light;
    Int XOMAP_Touch;
    Int XOMAP_TouchEx;
    IAUDDEC_Params;
}

TypeDef struct IOMAP_Status {
    Int XOMAP_Status;
    IAUDDEC_Params;
}

TypeDef struct expressDSP {
    Int size;
    Int XDSP_Handle;
    Int XDSP_Rx;
    Int XDSP_Tx;
    IAUDDEC_Params;
}
Define Additional Memory Blocks

Define DSP Algorithm Function

- xDM algos 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;
Ready to “Generate Code”

Final Step…
Click on the “Generate Code” button!

View Code Written by Component Wizard
Component Wizard Made Instance Object

/*
//==============================================
// 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()

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));
}
```
(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, eg: size of arrays, length of buffers, Q of filter, etc…

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Defined by</th>
<th>Allocated by</th>
<th>Written to by</th>
<th>Read from by</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof()</td>
<td>Algorithm</td>
<td></td>
<td></td>
<td>Algorithm</td>
</tr>
<tr>
<td>filterType</td>
<td>(in header file)</td>
<td>Application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filterOrder</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bufferSize</td>
<td>Algorithm</td>
<td>Application</td>
<td>Application</td>
<td>Algorithm</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Defined by: *Algorithm* (in header file)

Allocated by: *Application*

Written to by: *Application*

Read from by: *Algorithm*

Param Structures Defined in IMOD.H

```c
// 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;
```
# IFIR_Params : IFIR.C

```c
#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

```c
#include "ifir.h"

IFIR_Params IFIR_params;

IFIR_params = IFIR_PARAMS;

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

<table>
<thead>
<tr>
<th>size</th>
<th>alignment</th>
<th>space</th>
<th>attrs</th>
<th>*base</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Defined by:** IALG Spec & Algorithm (rtn 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...

<table>
<thead>
<tr>
<th>*fxns</th>
<th>filterLen</th>
<th>blockSize</th>
<th>*coeffs</th>
<th>*workBuf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Defined by:** Algorithm
- **Allocated by:** Application
- **Written to by:** Algorithm
- **Read from by:** Algorithm (private structure!)
The vTab Concept and Usage

```c
#include <ialg.h>
typedef struct IFIR_Fxns {
    IALG_Fxns ialg; /* 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

Pragmas - For Linker Control of Code Sections

```c
#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 algorithmic 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

**XDAIS Instance**

- **memTab**
- **params**
- **algo**
- **code**
- **vTab**
- **coeff[32]**
- **x[1031]**
- **x**
- **EX_TI_Obj**
- **EX_TI_Fxns**
- ***a**
- ***x**
- **handle**

---

**Multiple Instances of an Algorithm**

- **instObj1**
  - **fxns**
  - **a**
  - **tmp**
  - **hist**
  - **handle1**
- **instObj2**
  - **fxns**
  - **a**
  - **tmp**
  - **hist**
  - **handle2**

- **vTab**
- **Coeffs**
- **Scratch1**
- **history1**
- **history2**

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**

<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.

```
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++) {
   memTab[i].base = (Void *)memalign(memTab[i].alignment, memTab[i].size);  // Allocate memory for algo
   memTab[i].base = (Void *)memalign(memTab[i].alignment, memTab[i].size);  // Allocate memory for algo
}

3. alg = (IALG_Handle)memTab[0].base;
   alg->fxns = &fxns->ialg;  // Set up handle and *fxns pointer

4. fxns->ialg.algInit(alg, memTab, NULL, (IALG_Params *)params);  // Initialize instance object
```

```
1. IALG_MemRec memTab[1];  // Create table of memory requirements
   int buffer[15];  // Reserve memory for instance object

2. memTab[0].base = buffer0;  // with 1st element pointing to object itself

3. ISINE_Handle sineHandle;
   sineHandle = memTab[0].base;
   sineHandle->fxns = &SINE_TTO_ISINE;  // Create handle to InstObj
   sineHandle->fxns->ialg.algInit(IALG_Handle)sineHandle,memTab,NULL,(IALG_Params *)&sineParams);  // Setup handle to InstObj

4. sineHandle->fxns->ialg.algInit(IALG_Handle)sineHandle,memTab,NULL,(IALG_Params *)&sineParams);  // Set pointer to algo functions
```
Using EDMA3 and ACPY3

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>mem2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
</tr>
</tbody>
</table>

**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**

- DMA (sync)
- QDMA (async)
- EDMA3 (System DMA)
- C64x+ DSP

**Drivers**

- DMA
  - Enhanced DMA (version 3)
  - DMA to/from peripherals
  - Can be sync’d to peripheral events
  - Handles up to 64 events

**Both Share** (number depends upon specific device)

- 128-256 Parameter RAM sets (PARAMs)
- 64 transfer complete flags
- 2-4 Pending transfer queues

**QDMA**

- Quick DMA
- DMA between memory
- Async – must be started by CPU
- 4-8 channels available

**Algo's**

- DMA to/from peripherals
- Can be sync’d to peripheral events
- Handles up to 64 events
DMA : Basics

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?**

Let's look at a simple example...

```
Transfer Configuration

<table>
<thead>
<tr>
<th>Options</th>
<th>Source</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link Addr</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cnt Reload</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B Count (#Elements)</th>
<th>A Count (Element Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C Count (#Frames)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>
```
DMA Examples

EDMA Example : Simple (Horizontal Line)

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

Is there another way to set this up?

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>8</td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td>14</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td>20</td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td>26</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 frame size: 4 bytes
- SRCBIDX now will be 6 – skipping to next column
- DSTBIDX now will be 2

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>BCNT</td>
<td>SRCBIDX</td>
</tr>
<tr>
<td>ACNT</td>
<td>DSTBIDX</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>= &amp;loc_8</td>
<td>= &amp;myDest</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>BCNT</td>
<td>SRCBIDX</td>
</tr>
<tr>
<td>ACNT</td>
<td>DSTBIDX</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>= &amp;loc_8</td>
<td>= &amp;myDest</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 2 3 4 5 6</td>
<td>8</td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td>9</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td>10</td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td>11</td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td>14</td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td>15</td>
</tr>
</tbody>
</table>

- ACNT is defined here as 'short' element size: 2 bytes
- BCNT is again frame size: 4 elements
- CCNT now will be 5 – as there are 5 frames
- SRCCIDX skips to the next frame
**EDMA Example: Block Transfer (more efficient)**

**Goal:**
Transfer a 5x4 subset from `loc_8` to `myDest`

### 16-bit Pixels

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>16-bit Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

### myDest:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>
| 14| 15| 16| 17| 18|...

- **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** = &`loc_8`
- **Destination** = &`myDest`
- (4*2) is 8 = **DSTBIDX** = 12 is (8*2) (from block 8 to 14)
- 0 = **DSTCIDX** = 0
- **SRCCIDX** = 0
- **CCNT** = 1
Basic ACPY3 (memcpy on steroids)

ACPY3 Method Chronology

ACPY3_init()
Read to do ACPY3 work

ACPY3_configure()
Describe DMA actions to perform
ACPY3_start()
Begin DMA work
ACPY3_wait()
Optional CPU jobs in // w DMA
Spin until DMA completes

ACPY3_fastConfigure16()
Tweak for next DMA actions to perform
ACPY3_start()
Begin DMA work
ACPY3_wait()
Optional CPU jobs in // w DMA
Wait until DMA is done

ACPY3_exit()
No more ACPY3 work to follow

ACPY3_configure

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>Options</th>
<th>transferType</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcAddr</td>
<td>1D1D, 1D2D, 2D1D or 2D2D</td>
</tr>
<tr>
<td>dstAddr</td>
<td></td>
</tr>
<tr>
<td>elementSize</td>
<td>ACNT</td>
</tr>
<tr>
<td>numElements</td>
<td>BCNT</td>
</tr>
<tr>
<td>numFrames</td>
<td>CCNT</td>
</tr>
<tr>
<td>srcElementIndex</td>
<td>SRCBIDX</td>
</tr>
<tr>
<td>dstElementIndex</td>
<td>DSTBIDX</td>
</tr>
<tr>
<td>srcFrameIndex</td>
<td>SRCCIDX</td>
</tr>
<tr>
<td>dstFrameIndex</td>
<td>DSTCIDX</td>
</tr>
<tr>
<td>waitId</td>
<td>-1 (discussed later)</td>
</tr>
</tbody>
</table>

- srcId -
**DMA : Basics**

**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 8 9 10 11 12</td>
<td>8 9 10 11</td>
</tr>
</tbody>
</table>

**ACPY3_TransferType** | **transferType** | **ACPY3_1D1D** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Void* srcAddr</td>
<td>(IDMA3_AdrPtr) loc_8</td>
<td></td>
</tr>
<tr>
<td>Void* dstAddr</td>
<td>(IDMA3_AdrPtr) myDest</td>
<td></td>
</tr>
<tr>
<td>MdUns elementSize</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MdUns numElements</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MdUns numFrames</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MdInt srcElementIndex</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MdInt dstElementIndex</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MdInt srcFrameIndex</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MdInt dstFrameIndex</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MdInt waitId</td>
<td>-1 (discussed later)</td>
<td></td>
</tr>
</tbody>
</table>

**ACPY3 configure Example Code**

**Goal:**
Transfer 4 elements from loc_8 to myDest

```c
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_fastConfigure32b</td>
<td>Modify a single (32-bit) Parameter of the logical DMA 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_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

```c
void ACPY3_fastConfigure32b (IDMA3_Handle handle, ACPY3_PaRamField32b fieldId, unsigned int value, short transferNo);
```

```c
typedef enum ACPY3_PaRamField32b {
    ACPY3_PaRamFIELD_SRCADDR =   4,
    ACPY3_PaRamFIELD_DSTADDR = 12,
    ACPY3_PaRamFIELD_ELEMENTINDEXES = 16,
    ACPY3_PaRamFIELD_FRAMEINDEXES = 24
} ACPY3_PaRamField32b;
```

#### List of field Id's:

- **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.**

```c
typedef enum ACPY3_PaRamField16b {
    ACPY3_PaRamFIELD_ELEMENTSIZE =   8,
    ACPY3_PaRamFIELD_NUMELEMENTS = 10,
    ACPY3_PaRamFIELD_ELEMENTINDEX_SRC = 16,
    ACPY3_PaRamFIELD_ELEMENTINDEX_DST = 18,
    ACPY3_PaRamFIELD_FRAMEINDEX_SRC = 24,
    ACPY3_PaRamFIELD_FRAMEINDEX_DST = 26,
    ACPY3_PaRamFIELD_NUMFRAMES = 28
} ACPY3_PaRamField16b;
```
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

```c
#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);
```
Working with Fixed Resources

EDMA3 : Shared Resources

- ACPY3 uses the QDMA to perform transfers
- Up to 8 channels are shared among all algo's (limits us to 8 concurrent xfrs) *
- PARAMs and TCCs (complete flags) are shared by both the DMA & QDMA, as well as between algorithms
- We need a means to:
  - Allocate DMA resources
  - Share DMA resources

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><code>x = malloc(size); /*allocate*/</code></td>
<td>Create</td>
</tr>
<tr>
<td><code>x={...}; /*initialize*/</code></td>
<td>Get DMA resource requirements</td>
</tr>
<tr>
<td><code>filter(...); /*execute*/</code></td>
<td>Initialize DMA</td>
</tr>
<tr>
<td><code>free(a); /*release*/</code></td>
<td>Process</td>
</tr>
<tr>
<td><code>Use DMA Resources</code></td>
<td>Delete</td>
</tr>
<tr>
<td><code>Release DMA resource</code></td>
<td></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 ACPY3:
  - idMA3 implemented by algorithm authors
  - DMAN3 called by application authors
We've seen ACPY3, what about the other two? Let's start with iDMA3...
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>Create (&quot;Constructor&quot;)</td>
<td>algNumAlloc, algAlloc, algInit</td>
<td>dmaGetChannelCnt, dmaGetChannels, dmaInit</td>
</tr>
<tr>
<td>Process</td>
<td>algActivate, doDSP (i.e. VIDDEC_process), algDeactivate</td>
<td>ACPY3_activate, ACPY3_config, ACPY3_start, ACPY3_deactivate</td>
</tr>
<tr>
<td>Delete (&quot;Destructor&quot;)</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); }```

DaVinci/OMAP Workshop - Using EDMA3 and ACPY3
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)**

**DMAN3**

- **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

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

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
served concurrently by the
EDMA engine
(Optional) Introduction to DSP/BIOS Link

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.
(Optional) Introduction to DSP/BIOS Link

Introduction
Where Have We Seen DSP/BIOS Link Before?
What is DSP/BIOS Link?
Codec Engine Advantages
Which Should I Use – Codec Engine vs DSP/BIOS Link?
Guidelines
Use Cases
DSP/BIOS Link Architecture
DSP/BIOS Link Modules
PROC
MSGQ
POOL
CHNL
(Optional) RINGIO
What’s Next
How DSPLink API’s are used by Codec Engine
For More Information
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.

CE Framework - Review

Codec Engine (CE) uses DSP/BIOS Link for inter-processor communication (ipc)
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.

### What is DSP/BIOS™ LINK?

- **Lower level** inter processor communication link
- Allows master processor to **control execution** of slave processor
  - Boot, Load, Start, Stop the DSP
- Provides **peer-to-peer protocols** for communication between the multiple processors in the system
  - Includes **complete protocols** such as MSGQ, CHNL, RingIO
  - 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
- Provides **generic APIs** to applications
  - Platform and OS independent
- Provides a **scalable** architecture, allowing system integrator to choose the optimum configuration for footprint and performance

### DSPLink Features

- **Hides platform/hardware** specific details from applications
- **Hides GPP operating system** specific details from applications, otherwise needed for talking to the hardware (e.g. interrupt services)
- Applications written on DSPLink for one platform can directly work on other platforms/OS combinations requiring **no or minor changes** in application code
- Makes applications **portable**
- Allows **flexibility** to applications of choosing and using the most appropriate high/low level protocol
Codec Engine Advantages

**CE Framework**

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

**Use Case #1**

**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
  (next slide)

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

Is algorithm xDAIS compliant?

Using more than one algorithm?

Yes

No

Yes

No

DSPLink

Codec Engine
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
  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

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 omap356</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/OP3</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
DSP/BIOS Link Modules

- **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
  [http://wiki.davincidsp.com/index.php/Writing_DSPLink_Application_using_PROC_read_and_write_APIs](http://wiki.davincidsp.com/index.php/Writing_DSPLink_Application_using_PROC_read_and_write_APIs)
### 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, ...)

Message Queue "DSPMSGQ1"

This sequence is also valid for Message Queue created on GPP-side
### 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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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

GPP

- POOL_open()
- POOL_alloc(256, ...)
- POOL_free(buf, ...)
- POOL_close()

DSP

- POOL_open()
- POOL_alloc(1024, ...)
- POOL_free(buf, ...)
- POOL_close()
## POOL APIs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POOL_open</strong></td>
<td>Opens a specified pool referenced by the pool ID</td>
</tr>
<tr>
<td><strong>POOL_close</strong></td>
<td>Closes a specific pool</td>
</tr>
<tr>
<td><strong>POOL_alloc</strong></td>
<td>Allocates a buffer of specified size from a pool</td>
</tr>
<tr>
<td><strong>POOL_free</strong></td>
<td>Frees a buffer into the specified pool</td>
</tr>
<tr>
<td><strong>POOL_translateAddr</strong></td>
<td>Translates addresses between two address spaces for a buffer that was allocated from the pool</td>
</tr>
<tr>
<td><strong>POOL_writeback</strong></td>
<td>Writes the content of GPP buffer into DSP buffer (with offset in sync)</td>
</tr>
<tr>
<td><strong>POOL_invalidate</strong></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

![Diagram of CHNL](image)

*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

Reader
Writer

Writer acquires and releases data
to fill the buffer

Reader and writer both
acquire and release data

Empty Buffer
Full Buffer

Writer acquires
and releases data

Reader

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_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>
<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_getEmptyAttrSize</td>
<td>Returns the current empty attribute buffer size</td>
</tr>
<tr>
<td>RingIO_getEmptySize</td>
<td>Returns the current empty data buffer size</td>
</tr>
<tr>
<td>RingIO_getValidAttrSize</td>
<td>Returns valid attributes size in the RingIO</td>
</tr>
<tr>
<td>RingIO_getValidSize</td>
<td>Returns valid size in the RingIO</td>
</tr>
<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_setvAttribute</td>
<td>Gets an attribute with a variable sized payload from the attribute buffer</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_setNotifier</td>
<td>Sets Notification parameters for the RingIO client</td>
</tr>
<tr>
<td>RingIO_flush</td>
<td>Flush the data from the RingIO</td>
</tr>
</tbody>
</table>

RINGIO APIs contd..

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RingIO_getvAttribute</td>
<td>Sets an attribute with a variable sized payload at the offset provided in acquired data buffer</td>
</tr>
<tr>
<td>RingIO_getAttribute</td>
<td>Gets an attribute with a variable sized payload from the attribute 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_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

**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
- POOL_open
- MSGQ_TransportOpen
- MSGQ_alloc

- PROC_attach
- PROC_start
- MSGQ_create
- MSGQ_locate

- PROC_stop
- PROC_detach
- POOL_close
- MSGQ_TransportClose
- MSGQ_release

**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

**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 5

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?

   \[ \text{ioctl(fd, SNDCTL_DSP_CHANNELS, &numchannels)} \]
   \[ \text{ioctl(fd, SNDCTL_DSP_SPEED, &samplerate)} \]
   \[ \text{and they’re found in init_sound_device() } \]

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 be allocated by the driver. The driver then uses VIDIOC_QUERYBUFS 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.
1. In lab08b_audio_video_rtime, main.c, what priority is the audio thread set to? (You may use an expression here.)
   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.)
   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 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 XDPCPATH 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 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 –
CL  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
DSDK  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
Acronyms

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
Chapter Topics

Building Programs with the XDC Build Tool

XDC Overview ................................................................. 99-3
Packages (and Repositories) ......................................... 99-4
  package.xdc ............................................................... 99-5
  Repositories ............................................................... 99-6
Invoking XDC ............................................................... 99-8
XDC Build Flow ............................................................ 99-10
  config.bld ................................................................. 99-10
  package.bld .............................................................. 99-12
  package.mak ............................................................ 99-15
XDC Summary ............................................................. 99-16
Lab 99 – Using XDC ....................................................... 99-17
  Lab Objectives .......................................................... 99-17
  Lab99a_hello_world .................................................. 99-17
  Lab99b_extended_features ........................................ 99-22
  Questions .................................................................. 99-27
  Challenges ............................................................... 99-28
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)

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.
package.xdc

- "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...
Repositories

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.

Package and Repository (Example 1)

Repository: `/home/user/workshop/lab`
Package name: `app`

```
workshop
  lab
    app
      package
      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.

Three repo/pkg examples...
**Package and Repository (Example 2)**

**Example 2**
- Repository: `/home/user/workshop/lab`
- Package names: `app`, `codecs`
- Delivered as: `app.tar`, `codecs.tar`

```
workshop
  lab
    app
    codecs
```

**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`

```
dvevm
  codec_engine
    packages
      ti
        sdo
          ce
            video
```
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)**

```
#!/bin/sh
# Import installation paths
./setpaths.sh
# Define search paths
export XDCPATH="$CE_INSTALL_DIR/packages; $DVEVM_INSTALL_DIR; $SDK_INSTALL_DIR; $BIOS_INSTALL_DIR"
# Define options for execution
export XDCBUILDCFG="$(pwd)/../config.bld"
```

```
#!/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

Invoking XDC

XDC Invocation (Implicit Arguments)

Explicit Arguments

runxdc.sh

#!/bin/sh

# Import installation paths
. ../../../setpaths.sh

# Define search paths
export XDCPATH="$CE_INSTALL_DIR/packages; $DVEVM_INSTALL_DIR; $SDK_INSTALL_DIR; $BIOS_INSTALL_DIR"

# Define options for execution
export XDCBUILDCFG="$(pwd)/../config.bld"

xdc

Explicit Arguments

runxdc.sh

#!/bin/sh

# Import installation paths
. ../../../setpaths.sh

# Define search paths
export XDCPATH="$CE_INSTALL_DIR/packages; $DVEVM_INSTALL_DIR; $SDK_INSTALL_DIR; $BIOS_INSTALL_DIR"

# Define options for execution
export XDCBUILDCFG="$(pwd)/../config.bld"

xdc $@ -PR .

Pass all runxdc.sh args

build all pkgs recursively

Some Target Options

Some Target Options

Explicit Arguments

runxdc.sh

#!/bin/sh

# Import installation paths
. ../../../setpaths.sh

# Define search paths
export XDCPATH="$CE_INSTALL_DIR/packages; $DVEVM_INSTALL_DIR; $SDK_INSTALL_DIR; $BIOS_INSTALL_DIR"

# Define options for execution
export XDCBUILDCFG="$(pwd)/../config.bld"

xdc $@ -PR .

Pass all runxdc.sh args

build all pkgs recursively

Specifying XDC Targets

Implicit Arguments

Along with the explicit arguments XDC has three implicit arguments it looks for:

1. package.xdc

Names the package to be built and defines its contents

2. package.bld

Provides package specific build instruct’s

XDC allows for various targets...
XDC Build Flow

config.bld

**XDC Flow : config.bld**

<table>
<thead>
<tr>
<th>Invoke XDC</th>
<th>XDC <code>&lt;files&gt; &lt;XDCPATH&gt; &lt;XDCBUILDCFG&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Def's</td>
<td>config.bld (path specified by xdcbuildcfg)</td>
</tr>
</tbody>
</table>

- **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.

```plaintext
config.bld

Parts of config.bld:
- DSP Target
- ARM Target
- Linux Host Target
- Build Targets

/*
 * ======== 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)
config.bld

Parts of config.bld:

- DSP Target
- ARM Target
- Linux Host Target
- Build Targets

// ======== 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
- Target's 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)

config.bld

Parts of config.bld:

- DSP Target
- ARM Target
- Linux Host Target
- Build Targets

// ======== 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";
config.bld

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 xdbuildcfg)</td>
</tr>
<tr>
<td>Package Def's</td>
<td>packagexdc (current directory)</td>
</tr>
<tr>
<td>Executable/Library Build definitions</td>
<td>package.bld (current directory)</td>
</tr>
<tr>
<td></td>
<td>addExecutable</td>
</tr>
<tr>
<td></td>
<td>...</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, ...);
```

Looking more closely at `addExecutable`...

```javascript
Pkg.addExecutable("app_debug", MVArm9, MVArm9.platform, {
  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:
MVArm9 with debug profile

```javascript
Pkg.addExecutable ("myApp_debug", MVArm9, MVArm9.platform, {
  cfgScript: null,
  profile: "debug"
}).addObjects ( ["main.c"] );
```

### 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

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)</td>
</tr>
<tr>
<td></td>
<td>addExecutable</td>
</tr>
</tbody>
</table>
| | ...
| Generated by XDC | package.mak (internal XDC script to create) |

- 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

(setq)
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**

- 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.

```bash
# 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):

```
<table>
<thead>
<tr>
<th>Workshop Files (common to all lab exercises)</th>
</tr>
</thead>
<tbody>
<tr>
<td>config.bld</td>
</tr>
<tr>
<td>setpaths.sh</td>
</tr>
<tr>
<td>lab99a_hello_world</td>
</tr>
<tr>
<td>runxdc.sh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>app (found in <code>lab99a_hello_world</code> app directory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
</tr>
<tr>
<td>package.bld</td>
</tr>
<tr>
<td>package.xdc</td>
</tr>
</tbody>
</table>
```

5. Examine the C source file `app/main.c` which prints the string “hello world” to standard output.

```bash
# 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:

```bash
# 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:

```bash
# 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:

```
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`.

```makefile
Pkg.makeEpilogue += "include ../../custom.mak\n\n"
```

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!