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 ©2008 by Texas Instruments Incorporated. All rights reserved.

Technical Training Organization
Semiconductor Group
Texas Instruments Incorporated
7839 Churchill Way, MS 3984
Dallas, TX  75251-1903

Revision History

1.0-1.2    8/99 – 01/00
2.0-2.7    5/00 – 10/03
3.0-3.4    3/05 – 06/07
4.0        4/08
Introduction

Welcome to the Texas Instruments DSP/BIOS workshop.

In this chapter an overall outline of the class is provided.

Objectives

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

- Define key software design challenges in developing real-time systems
- Demonstrate essential skills in the use of Code Composer Studio (CCS) in authoring a real-time system
- Identify and apply the optimal DSP/BIOS constructs to implement a given real-time system
- Analyze and optimize a software solution to meet real-time requirements

Module Topics

Introduction ............................................................................................................................................... 1-1
Workshop Agenda .................................................................................................................................... 1-2
EVM Overview ......................................................................................................................................... 1-3
Lab – System Setup ................................................................................................................................. 1-4
A. Computer Login ............................................................................................................................ 1-4
B. Connecting the EVM to the PC ...................................................................................................... 1-5
C. CCS Setup ....................................................................................................................................... 1-6
D. Setup CCS Options ......................................................................................................................... 1-7
Workshop Agenda

The outline for this workshop is as listed below. Four to five modules should be covered each day, although the actual pace of the class will vary based on the interests and needs of the class.

<table>
<thead>
<tr>
<th>DSP/BIOS Workshop - Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>2. Real-Time System Design Considerations</td>
</tr>
<tr>
<td>3. Hardware Interrupts (HWI)</td>
</tr>
<tr>
<td>4. Software Interrupts (SWI)</td>
</tr>
<tr>
<td>5. Task Authoring (TSK)</td>
</tr>
<tr>
<td>6. Data Streaming (SIO)</td>
</tr>
<tr>
<td>7. Multi-Threading (CLK, PRD)</td>
</tr>
<tr>
<td>8. BIOS Instrumentation (LOG, STS, SYS, TRC)</td>
</tr>
<tr>
<td>9. Static Systems (GCONF, TCONF)</td>
</tr>
<tr>
<td>10. Dynamic Systems (MEM, BUF)</td>
</tr>
<tr>
<td>11. Inter-Thread Communication (MSGQ, ...)</td>
</tr>
<tr>
<td>12. Input Output Mini-Drivers (IOM)</td>
</tr>
<tr>
<td>13. DSP Algorithm Standard (XDAIS)</td>
</tr>
<tr>
<td>14. Reference Frameworks (RF1, 3, 5, 6)</td>
</tr>
<tr>
<td>15. Review</td>
</tr>
</tbody>
</table>

Outside the scope of this workshop are topics such as:
- DSP Theory – textbooks are suggested in the final module
- Processor architectures - taught in C55x or C6x Workshops and via TI technical publications
- Operating System Theory or authoring

While not required, the following prerequisites are recommended for those considering this workshop:
- Familiarity with coding in C language
- Experience with software development and programming methodologies
- Familiarity with CCS development tool
- Helpful - familiarity with:
  - C6x or C5xx Processor Architectures
  - Object-oriented programming methodologies
EVM Overview

**TMS 320 DM 6437 EVM**

- **DIP Switches**
- **LEDs**
- **Line In**
- **USB Port**
- **Power**
- **Headphone**

---

**EVM Resets**

- **CCS reset**
  - Use most commonly – fast and easy
  - Invoked via: Debug -> DSP Reset
  - Resets DSP (not full board)
  - May not clear all states required for ‘clean’ new debug session

- **Reset button**
  - More extensive reset operation, still not comprehensive
  - OK to assert when CCS (3.1 or higher) is running

- **Absolute reset**
  - Provides completely ‘fresh’ starting point
  - remove Power and USB plugs
  - Best choice to be sure a full reset is obtained
Lab – System Setup

A number of different Evaluation Modules (EVMs) and DSP Starter Kits (DSKs) can be driven by Code Composer Studio (CCS). This first lab exercise will provide familiarity with the method of testing the hardware and setting up CCS to use the selected target. Steps in this lab will include those noted in the diagram below:

### Lab 1 - Objectives

**Software**
- Run CCS Setup
- Start CCS
- Configure CCS Options
- Component Manager
- Close CCS

**Hardware**
- Hook up the EVM
- Supply power

Time: 20 minutes

### A. Computer Login

1. If the computer is not already logged-on, check to see if the log-on information is posted. If not, please ask the instructor (student/student is a common ID/psw to try).
B. Connecting the EVM to the PC

The software should already be installed on the lab workstation. All that should have to be done is to physically connect the EVM.

2. **Connect a USB cable between the EVM’s USB port and a USB port on the PC.**
   
   If you connect the USB cable to a USB Hub, be sure the hub is connected to the PC or laptop and power is applied to the hub).

   **Note:** Note: If, after plugging in the USB cable, a found new hardware message appears indicating that the USB driver needs to be installed, notify your instructor. In most classroom installations, this has already been performed.

3. **Plug in the audio cables:**
   
   − Use a stereo mini plug to connect the PC audio line out to the EVM audio **LINE IN**.
   
   − Use another stereo mini plug to connect the EVM **HP OUT** to the headphones/speaker.
     
     Do *not* connect to the line out, as one of the drivers used does not send signal out to them.

   Assure that the plugs are fully inserted so that the audio will be reliably transferred.

4. **Plug the power cord of the power supply into an AC source.**

   The power cable must be plugged into AC source prior to plugging the 5 Volt DC output connector into the EVM.

5. **Plug the power supply output cable into the EVM’s power receptacle.**

   When power is applied to the board, the Power-On Self-Test (POST) will run. LEDs DS501 (next to the USB plug) will light briefly, flicker and go off. LEDs DS502 and DS5 will remain on. Do not turn on CCS until DS501 goes off.

   **Note:** At this point, if you were installing the EVM for the first time on your own PC you would now finish the USB driver installation. This step has already been performed on the workshop PCs.
C. CCS Setup

Code Composer Studio (CCS) supports numerous TI processors (including the C6000 and C5000 series) and a variety of target boards (simulators, EVMs, DSKs, and XDS emulators). The CCS Setup utility is used to select the device and board for CCS to work with. For this workshop, the C6416 DSK-USB will be chosen.

1. **Start CCS Setup:** by double-clicking the Setup icon on the PC desktop:
   Be aware there are two CCS icons, one for setup, and the other run CCS itself. Here, the **Setup CCStudio v3.3** icon is the one to select. Note: the program can be run directly from: `C:\CCStudio_v3.3\cc\bin\cc_setup.exe`

   When CC_Setup opens, a screen similar to this should appear:

   ![CCS Setup Screen](image)

2. **Clear the previous configuration:** If a previous configuration exists, clear it by selecting the **remove all** button in the bottom right hand corner of the left pane. (If no configuration is currently loaded, this button will be greyed out as above.) Confirm when prompted. When finished, the left pane of the setup program should look as above, with nothing listed underneath the MySystem icon.

3. **Pick the desired target:** Select the **DM6437** in the Import Configuration box and click the **<<Add** button. Your options for selecting a board configuration appear in the middle pane. If many board configuration options appear here, use the filtering options to reduce the number. In the following screen, “EVM” was selected from the Platform pulldown list at the top of the pane. The “<< Add” button is located in the lower left corner of the middle pane. Note: there are two DM6437 EVM choices. Click on each and observe their file names. Select the one that does not have _v2 in the file name.

4. **Lock in choices:** Click on **Save and Quit**, select **Yes** when prompted to start CCS on exit.
D. Setup CCS Options

To assure an efficient lab environment, a few CCS options will now be verified and/or set.

1. Component Manager: A new feature of CCS is the ability to easily choose which release of DSP/BIOS to use. CCS 3.3 ships with BIOS 5.31.02 To upgrade to BIOS 5.31.08, download it via the update advisor (already done here) and apply it via Help | About | Component Manager. Open the Target Content and TMS320C64XX folders and check BIOS 5.31.08. Close the window, and as prompted, close and restart CCS to make the change complete.

2. Modify editor properties: The editor’s properties may be accessed from the Options pulldown via Option | Color | Editor Color…, Option | Font | Editor Font…, and Option | Editor | Properties… As an example, we will modify the properties of the Editor’s demarcation of comments.
   - Begin by selecting Option | Color | Editor Color…
   - In the colors tab (which should already be open), click on comments in the Window Text section.
   - Uncheck the Italic check-box amongst the Font style options to the lower right of the dialog box.
   - Click OK to close this dialog box.

3. Set the properties of the Debugger. Select Option | Customize… and click on the Debug Properties tab (the leftmost tab). You should see a window like the one below.

4. Select the options that you would like for the debugger. The following are recommended:
   - Uncheck Open the Disassembly Window automatically
   - Check Perform Go Main automatically
   - Check Connect to the target at startup
   - Check Remove remaining debug state at connect
   - Leave other options at their default values
5. Specify the Program Load Options: Move to the **Program/Project CIO** tab. The recommended options are shown to the right. *Load program after build* automates a step otherwise requiring the use of *File → Load Program* to load newly built projects.

6. Specify the desired CCS Title Bar Properties via the **Control Window Display** tab: To reach this tab may require using the scrolling arrows at the end of the tab display) as show below. By selecting “**Board Name**”, “**Current Project**”, “**Currently loaded program**” and “**Display Full Path**” the CCS title bar will specify all the key information on what CCS is currently set up to do. The last two selections under “**Project close**” allow for a fast and clean end of a session.

7. Select **OK** to close the CCS Customization window.

8. **Exit Code Composer Studio.**
Real-time System Considerations

Introduction

In this chapter an introduction to the general nature of real-time systems and the various goals in creating a system will be considered. Each of the concepts noted here will be studied in greater depth in succeeding chapters.

Objectives

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

- Describe the topology of most common DSP systems
- Identify several competing goals in the design of a DSP system
- List various thread types and note some key characteristics of each
- Describe the basics of object based programming
- Create and debug a simple DSP project using CCS

Module Topics

Real-time System Considerations ............................................................................................................ 2-1

DSP System Topology .......................................................................................................................... 2-2
DSP/BIOS Conventions ....................................................................................................................... 2-8
Lab 2: CCS Skills .................................................................................................................................... 2-10
A. Project Management ..................................................................................................................... 2-11
B. Debugging Techniques ................................................................................................................ 2-14
C. (Optional) Authoring Skills ............................................................................................................. 2-16
D. (Optional) BIOS Instrumentation .................................................................................................. 2-17
CCS Reference Sheet ......................................................................................................................... 2-19
DSP System Topology

Definitions / Vocabulary

- Real-Time Systems: where processing must keep up with rate of I/O
- DSP/BIOS: a scalable, real-time kernel, used in thousands of systems today. Part of CCS, requires no license fees when used on TI DSPs
- Function: Sequence of program instructions that produce a given result
- Thread: Independent sequence of program instructions (functions) that execute within a specific context (registers, stack, priority).
- Object: a software bundle of variables and related methods
- Instance: a copy (or “channel”) of an object; implies reentrancy
- Reentrant: able to run multiple instances concurrently
- API: Application Programming Interface – methods for interacting with routines
- Events: cause a non-sequential change in the software flow of control
  - Synchronous Events: occur at predictable times (eg: if/else, polling)
  - Asynchronous Events: occur at unpredictable times (interrupts)
- Stream: large and frequent block of data; continuous
- Message: typically smaller and infrequent packets of data; sporadic

IPO

Input
CSL, HWI
Device Driver
DEV, IOM, PSP

Process
SWI - launch on I/O
TSK - post on I/O
XDAIS
algo

Output
CSL, HWI
Device Driver
DEV, IOM, PSP
Polling vs Interrupt (Event) Driven

Polling:
- Overhead of repeated checking
- Wastes MIPS, Watts
- Doesn’t allow other threads to run in the mean time

Interrupts:
+ No checking – launch on event
  + no wasted time or power
+ Allows other threads to run independently
+ Represent response to priority events
- Small number of interrupt sources to post ISRs

“Software” Interrupts and Semaphore posting
+ Allows interrupt/event launch of threads beyond ISRs
+ BIOS HWI & SWI are both posted to run, like an ISR
+ BIOS tasks (TSK) can be synchronized via SEMaphores
+ Improved Modularity
Hardware and Software Interrupt System

Execution flow for flexible real-time systems:

```
INT ! Hard R/T Process Post SWI Cleanup, RETURN HWI
           SWI Ready            Continue Soft R/T Processing ...

HWI
   - Fast response to interrupts
   - Minimal context switching
   - High priority for CPU
   - Limited number of HWI possible

SWI
   - Latency in response time
   - Context switch performed
   - Selectable priority levels
   - Execution managed by scheduler

DSP/BIOS provides for HWI and SWI management
DSP/BIOS allows the HWI to post an SWI to the ready queue
```

Thread Type Comparison

<table>
<thead>
<tr>
<th></th>
<th>HWI</th>
<th>SWI</th>
<th>TSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int x ... l ... P ... O ...</td>
<td>usually sample oriented</td>
<td>HWI : Input</td>
<td>SWI : Process</td>
</tr>
<tr>
<td></td>
<td></td>
<td>post</td>
<td>post</td>
</tr>
<tr>
<td></td>
<td>post</td>
<td>BUF</td>
<td>BUF</td>
</tr>
<tr>
<td></td>
<td>IOM (HWI)</td>
<td>SIO</td>
<td>IOM (HWI)</td>
</tr>
<tr>
<td></td>
<td>usually block oriented</td>
<td>usually block oriented</td>
<td>usually block oriented</td>
</tr>
<tr>
<td>Memory Size</td>
<td>HWI – RF1</td>
<td>SWI – RF3</td>
<td>TSK – RF5</td>
</tr>
<tr>
<td>Tiny</td>
<td>Small</td>
<td>Medium-Large</td>
<td></td>
</tr>
<tr>
<td>Response Time</td>
<td>ISR Rate</td>
<td>Quick</td>
<td>Slower</td>
</tr>
<tr>
<td>Reuse / Modularity</td>
<td>Minimal</td>
<td>Good</td>
<td>Excellent</td>
</tr>
<tr>
<td>Buffering Model</td>
<td>n/a</td>
<td>Ptr pass / SIO</td>
<td>SIO / ptr pass</td>
</tr>
<tr>
<td>Scheduler Dimensions</td>
<td>1 (HWI only)</td>
<td>2 (HWI,SWI)</td>
<td>3 (HWI,SWI,TSK)</td>
</tr>
</tbody>
</table>
Data Collection: Sample vs Buffers

Buffer management options

- **Single sample** – low latency, simple, easy
  - Usually best implemented in HWIs – less context switch overhead

- **Buffered data** – less frequent context switch overhead, better suited to multi-threaded systems. *Multiple* buffers (2 or more) required for sustained R/T performance (1 to collect new data while another is being processed)
  - Usually best if processing is in SWI or TSK to allow new blocks to be built via HWI
  - Single context switch for entire block keeps overhead to low/insignificant levels

<table>
<thead>
<tr>
<th>Data Processing</th>
<th>Single Sample</th>
<th>Multiple Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>&lt; 1 sample time</td>
<td>Up to total buffer depths</td>
</tr>
<tr>
<td>Context switch overhead</td>
<td>Data rate</td>
<td>Data rate / buffer size</td>
</tr>
<tr>
<td>Best for</td>
<td>Small systems</td>
<td>Larger systems</td>
</tr>
<tr>
<td></td>
<td>Control systems</td>
<td>Non-recursive systems</td>
</tr>
</tbody>
</table>

Single vs Double Buffer Systems

*Single buffer system: collect data or process data – not both!*

- One buffer can be processed while another is being collected
- When SWI/TSK finishes buffer, it is returned to HWI
- TSK is now ‘caught up’ and meeting real-time expectations
- HWI must have priority over SWI/TSK to get new data while prior data is being processed – standard in DSP/BIOS

*Double buffer system: process and collect data – real-time compliant!*

- Nowhere to store new data when prior data is being processed
Complex systems benefit from the availability of more sophisticated support, including:

- Multitasking scheduling allows lower priority threads to be preempted by higher ones
- TSKs encapsulate and abstract processing code. Each is assigned a desired priority
- Streams abstract the flow of data between drivers and processing threads (TSK, SWI)
- XDAIS and IOM standards:
  - Define known standards across all algos and drivers, respectively
  - Assure non-interference between components in complex systems
  - Offers system integrator a uniform interface to all code components

Modularity & Reuse

**Modularity**
- Don’t need to know everything
- Divide project by skills required
- Each segment is more manageable than the whole
- Easier to modify
- Easier to test
- More maintainable
- More adaptable for reuse

**Reuse**
- Maximize return on cost of SW development
- Product revisions
- Product range (standard/minimalist/deluxe versions)
- Standardized Interfaces
System Design Options and Tradeoffs

- **Dynamic vs Static** (Module 10)
  - Static systems – are smaller and faster code solutions, simpler to create and manage
  - Dynamic systems – allow blocks of RAM to be ‘borrowed’ from heap when needed, and returned afterward for reuse by subsequent requestors; add the create & delete phases

- **MIPS vs Mbytes** – system designer can often trade one for the other to optimize performance and cost

- **Number of Buffers**:
  - Latency (input to output time) vs flexibility (improved ability to tolerate preemption)

- **What is speed? MIPS vs TTM** (Time To Market)
  - Faster DSP processing rates offer performance that exceeds minimum requirements of many systems
  - More sophisticated features can be employed to simplify coding effort, improve speed of coding and time to market

- **Cost: Device vs TTM**
  - Price of DSP HW and development should be weighed against the value of time to market

---

**RTOS vs GP/OS**

<table>
<thead>
<tr>
<th></th>
<th>GP/OS</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>General</td>
<td>Specific</td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td>Large: 10K-20M</td>
<td>Small: 1K-20K</td>
</tr>
<tr>
<td><strong>Event response</strong></td>
<td>1ms to .1ms</td>
<td>100 – 10 ns</td>
</tr>
<tr>
<td><strong>File management</strong></td>
<td>FAT, etc</td>
<td>Not supported</td>
</tr>
<tr>
<td><strong>Dynamic Memory</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Threads</strong></td>
<td>Tasks, Ints</td>
<td>TSK, SWI, HWI</td>
</tr>
<tr>
<td><strong>Scheduler</strong></td>
<td>Preemption time slicing</td>
<td>Preemption only</td>
</tr>
<tr>
<td><strong>Host Processor</strong></td>
<td>ARM, x86, Power PC</td>
<td>DSP: C2000, '5000, '6000</td>
</tr>
</tbody>
</table>
DSP/BIOS Conventions

DSP/BIOS Environment

- DSP/BIOS is a library that contains a collection of modules with a particular
  - Interface and calling conventions
  - Set of data structures defined in the module’s header file
- Application Program interfaces (API) define the interacts with a module
  - Relates a set of constants, types, variables and functions visible to user programs
  - Object based: global parameters that control operation of each instance
- Objects - are structures that define the state of a component
  - Pointers to objects are called handles
  - References to the object are via the handle
  - Object based programming offers:
    - Better encapsulation and abstraction
    - Multiple instance ability

Module Interface

What is the advantage of object based programming
- Well defined interfaces
- Encapsulates and hides data
- Promotes code reuse
DSP/BIOS Naming Conventions

- Three or four letter module prefix naming convention
  - Used as a prefix for API, header files, and objects for modules
- Capitalization convention distinguishes functions, types, constants

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>CONVENTION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Calls</td>
<td>MOD_lowercase</td>
<td>LOG_printf</td>
</tr>
<tr>
<td>Data Types</td>
<td>MOD_Titlecase</td>
<td>LOG_Obj</td>
</tr>
<tr>
<td>Constants</td>
<td>MOD_UPPERCASE</td>
<td>TRC_USER0</td>
</tr>
<tr>
<td>Internal Calls</td>
<td>MOD_F_lowercase</td>
<td>FXN_F_nop</td>
</tr>
</tbody>
</table>

Note: Do not use internal calls

Data Types

DSP/BIOS standard data types & constants for portability

- **Arg**: generic argument type
- **Bool**: boolean value (TRUE or FALSE)
- **Byte**: minimal addressable unit
- **Char**: signed character value
- **Int**: signed integer value
- **Long**: signed long integer value
- **Uns**: unsigned integer value
- **Bits**: unsigned bit string
- **Ptr**: generic pointer value
- **Short**: signed short integer value
- **String**: pointer to character
- **Void**: empty type
- ...
Lab 2: CCS Skills

The primary purpose of this lab is to become familiar with CCS authoring and debugging techniques. Most of the skills exercised in this lab will be required to implement subsequent labs, so it is highly recommended that you take your time to absorb as many of these methods as possible, so that they will be skills available to you when needed later on. If desired, make notes as you go along on the job aid (“cheat sheet”) provided at the end of this lab. This sheet can be removed from the course notes and kept nearby as a reminder during later labs.

This lab is divided into four parts. The objective in part A of this lab is to acquire the skills needed to create and manage projects in CCS. In part B, a number of debugging methods will be explored. Part C provides an opportunity to learn the authoring techniques within CCS. In the final section of the lab, a brief preview of some advanced BIOS debugging tools will be provided.

This lab employs elements through lab 8, of sufficient complexity to showcase a variety of the techniques noted above. It is not expected that many of the constructs in these files will be understood at this point. Subsequent study and lab activity will build to a full understanding of them during the course of this workshop.
A. Project Management

1. **Launch CCS** (double click the CCS icon on the PC desktop).

2. **Open a starter project:** On the CCS Menu bar, select: Project | New and fill out the dialog box as per the graphic below. Click on the Finish box when completed.

   ![Project Creation Dialog](image)

   **Helpful:** Use the search icon at the right of the Location box to search for the desired path

   ![Helpful Tip]

   **Note:** A CCS project defines all the files in a program, the build options, and all other details necessary to author a system to run on a TI DSP.

3. **Add files to the project:** There are three techniques for adding files to projects:

   - Method 1: open **Window Explorer**; navigate to the directory where the source files reside; select any or all the source files to add the project, **drag and drop** these files on top of the project folder on the left pane in the CCS window.

   - Method 2: From the CCS menu bar, select: Project | Add Files to Project ...; navigate to the directory with the desired files; double-click on a file you want to add - or select all the files you want to add the project, then click the open button.

   - Method 3: **Right click on the project** in the Project Window and select **Add Files to Project**... This will call up a similar dialogue window to method 2.

   **Files to add to the project** (note: CCS automatically adds referenced header files)

   - From directory C:\BIOS\Labs\Work\
     - audio.c The main code for an audio filter application
   - From directory C:\BIOS\Labs\Algos\ add:
     - fir.c An FIR filter that does DSP filtering on the audio data
     - coeffs.c Tables of filter coefficients used by fir.c
     - nop_loop.asm An assembly program that implements a simple CPU load
     - load-2.c A ‘dummy load’ program, generic example of a 2nd thread
   - From directory C:\BIOS\Labs\HW\ add:
     - codec.c Program to setup and interact with serial ports and AIC33
4. **Add to the project a configuration (.tcf) file:** BIOS based projects employ a “textual configuration” file that defines the environment of a project: a listing of memory available in target hardware, how to route software elements to available memory, definitions of all the BIOS objects in use, interrupt routing, and so forth – all of which will be examined in subsequent chapters.

To specify a BIOS configuration for the project, two files need to be added:

- From directory C:\BIOS\Labs\Work\  
  - system.tcf The textual configuration file (can also be edited using GUI tool)
  - systemcfg.cmd Derived from the TCF file; specifies the linker procedure

5. **Specify the target DSP:** From the CCS pull-down menus, click on **Project | Build Options** ... select the Compiler tab and the Basic category. In the Target Version box, select C64x+ (-mv6400+). This is the CPU type of the 6437.

6. **Specify search paths:** In the Preprocessor category, type ..\HW; ..\Algos; $(BSL) into the Include Search Path box and close the dialog box by clicking on **OK**.

7. **Add library files to the project:** This project requires the Board Support Library (BSL), which are provided by Spectrum Digital with the EVM6437. There are two ways to add library files to the project:

   - **Method 1 - Drag and drop:** Navigate to the library file’s directory using windows explorer and drag the library file into the CCS project’s library folder. (recommended)

   - **Method 2 - Specify the include paths and add library files via linker tab:** Click on **Project | Build Options** ... select the Linker tab and place search paths in Library Search Path box, and library names in the Include Libraries box.

Add the following library:

```
evmdm6437bsl.lib from C:\CCStudio_v3.3\boards\evmdm6437_v2\lib\n```

8. **Expand the demo.pjt and Source folders** by clicking on the + to the left of the folder icons in the CCS Project view window. **Verify** that the project now **has the files as shown** to the right:

Note that you can right click on a file to view its directory source (under the properties option), double click on the file to open it onto the CCS workspace for viewing and editing, or delete it from the project (right click, delete).
9. Verify that the **Active Configuration** window displays **Debug** (and not **Release**). Debug mode provides full symbolic visibility for easy code review. Release mode, used later, invokes selected optimization, which often limits debug visibility. Since each has its own benefits, both are supported in any given project.

10. **Save the project**: via **Project | Save**

11. **Build the project**: via **Project | Build** or by clicking the icon.

   Note the progress of the build in the **Output window** at the bottom of CCS. When done, the results should automatically be loaded to the EVM target, as indicated by the **Loading Program** popup window that should briefly appear during the download. If errors are reported, recheck the steps above and try again. If there are still errors, ask the instructor for assistance.
B. Debugging Techniques

1. If audio.c is open with a yellow arrow at main(), the program is ready to run.

2. On the PC, begin playing music as the input to the EVM via the audio patch cable. Turn on the output speaker, or prepare to listen to the headphones.

3. Start the program running via Debug | Run, function key F5, or the run icon:

4. Verify that the selected music is now playing from the output device.

5. To enable/disable the FIR filter via CCS, open a watch window on the filter control: Open audio.c (double click on it in the Project View window); find and double-click on the variable sw0 to select it. Right click on the now highlighted variable and select Add to Watch Window. Change the value shown in the watch window from 1 to 0 to bypass the filter (and back to 1 to re-apply the filter function).

6. As an alternate way to add watch items: Drag and drop the variable sw1 onto the watch pane. Changing its value from 0 through 2 will allow differing filter functions to be applied. Optional: to peruse the coefficients, scan the file coeffs.c

7. Optional - Add a GEL file to the project: A GEL (general extension language) file is a CCS macro script that can automate keystrokes, add menu items, and visual controls – all of which are conveniences when debugging. To add the GEL, go to File | Load GEL... and navigate to C:\BIOS\Labs\Gels and select the file Control.gel. Go to the GEL menu and under Filter Controls experiment with the new functions now found there. On and Tone open slider controls that allow variables to be controlled by the dragging of the slider to a new setting. When trying this, observe the change in the sound as well as the corresponding changes in the variable values in the watch window.

8. Halt the program execution: To halt the running program type Debug | Halt, or function key Shift+F5, or the halt icon: After testing the halt function, resume the program by asserting run again.

9. Finding text in code: Locate the while loop in the procBuf code by typing <ctl>F or Edit | Find, then type in while. Pressing OK brings the display to the correct area and highlights the while(1) sought.

10. Set a breakpoint: Multiple breakpoints can be defined in the code listing, and the system will halt whenever a breakpoint is encountered. This is a very common and helpful debugging tool. Setting a breakpoint is very simple – click on any desired line (try the SEM_pend just under the while statement) and press F9. Note the confirmation of the breakpoint by the addition of the red dot on the left margin at the selected line. Run the code and observe the system halts at the selected breakpoint, as indicated by the yellow arrow on the red dot. Set another breakpoint or two in the while loop and run a few times more.

11. Run to cursor: When the code is halted, click on another line within the while(1) loop and select Debug | Run to Cursor or <ctl> F10 (or right-click and select Run to Cursor). This will cause the program to run until it encounters the line the cursor is currently at. If you like, try again at another location within the while loop. This ability is sometimes a convenient debugging control option.
12. **Stepping through code:** For even finer debug observation, the ability to run one line of code at a time is often required. This ability is provided by the Debug | Step Over, or – more conveniently – via the F10 key. Press F10 multiple times and observe the progress of the program as indicated by the yellow arrow in the left margin. Note: try to wait for the debugger to show HALTED: sw breakpoint in the lower left corner of CCS before issuing additional run commands.

13. **Running free of breakpoints:** A handy option when many breakpoints are set is the ‘run free’ mode, where breakpoints are bypassed without having to be cleared first. Try Debug | Run Free or <ctl>F5 to use run free mode.

14. **Clearing breakpoints:** To clear a breakpoint, simply click again on a line where a breakpoint is set and press F9 again. Notice the red dot disappears, indicating the clearing of the breakpoint. To clear all breakpoints, click  or Debug | Breakpoints (remove all).

15. **View memory:** It is often handy to be able to observe arrays in memory. CCS provides this ability via View | Memory. Specify &in (the label for the input buffer array) as the address, and for in the Format box at the bottom of the window, select Hex 16-Bit – TI Style. Make the window large enough to see 100 or more data values. Run to breakpoint a few more times and notice the values within the memory window updating at various times. Note: arrays can also be displayed in the watch window.

16. **Load a workspace file:** By now, the CCS workspace has probably become rather crowded. Time was expended making room for each new pane added. At this point, to start over, more time would be spent to reopen and rearrange the panes. To minimize this time CCS provides “workspaces”, which record all the project and display settings to a file which can be reloaded at any later point. This is a great convenience in the iterative world of debug, and their use is highly recommended. Use File | Workspace | Load Workspace and select Lab-2.wks from C:\BIOS\Labs\Wks. After a moment, the screen should be loaded with the windows described above arranged in a particular layout.

Note that the Memory view window changed appearance when the workspace was loaded. Originally, it was bound to the right edge of the CCS window, but now is floating in the main window. Most CCS panes can float in the main window, ‘dock’ to an edge, or be independent of the CCS main window. These options are set by right clicking on a given pane and manipulating the “Float in Main Window” and “Allow Docking” options. Experiment with a few such options as desired.

17. **Save a workspace:** Once the layout is to your liking, save the workspace as Lab02b.wks via File Workspace | Save Workspace As.

18. **Reloading a program:** If a file becomes corrupted during test, CCS offers the ability to reload the program via File | Reload Program.

19. **Save the project.** The C:\BIOS\Labs\Work directory will be the base path for all succeeding labs. So, if you want to keep a copy of your work for future study, it is recommended that you save the contents of the work directory after each lab. To facilitate this process, the directory C:\BIOS\mySols was created with an empty subdirectory to hold a copy of each of your lab results. Copy the contents of C:\BIOS\Labs\Work to C:\BIOS\mySols\02.
C. (Optional) Authoring Skills

In this lab, completed code was provided as a starting point. In subsequent labs, this will not be the case – the work required to implement a given solution will be a major point of all future labs. As such, knowledge of how to author C and other files in CCS will be required. In this section, key CCS authoring skills will be reviewed. The bulleted items below are for review and passive test only – do not save any work during these steps.

Basic Editor Commands:
- The editor in CCS shares some commands with most other text editors, such as Windows Notepad. The File commands to Open, Close, Save, and Save As, are probably already familiar to you. If not, familiarize yourself with those now.
- A slight difference is found with the File | New command – instead of opening a new C source file directly, there are options offered. The first option is for a New | Source File, also possible with the shortcut <ctl>N. The other important option is for a DSP/BIOS Configuration file. The ‘config’ file is a key component of BIOS-based projects. Via the config file, all system settings, such as the memory map, how to link software to memory, setup and assignment of interrupts, and static creation of all BIOS software objects is performed. The creation of a new BIOS configuration will be considered in the next chapter, and increasing use of configuration options will be a component of all future labs. For now, just note that this is the route to follow when a new config file is to be created. Also notice one extra command in the File menu – Save All: handy when locking in changes made to several files in a given project.
- Also similar to Notepad and other text editors are several Edit menu options and their shortcut keys: Cut (ctl-X), Copy (ctl-C), Paste (ctl-V), Select All (ctl-A), Delete (delete), Find/Replace (ctl-F), Undo (ctl-Z), Redo (ctl-Y).

Doing Some Editing in this Lab:

While the code seen here does currently run, a few changes can now be made to try out some of the skills covered above, and to observe a few additional editing support features.

1. Verify the current system: Rebuild <alt>P,B and verify that music will play as before <alt>D, R. If there are problems, you can retrieve a working copy of the lab from C:\BIOS\Sols\02. Once original performance has been verified, halt the code <alt>D, H.

2. Introduce an intentional error: In audio.c, locate the line reading: if( sw0 == 1 ). Delete the “0” from the variable name. What do you predict will happen when attempting to rebuild? Save the file, close the audio.c file window and rebuild the project. Note in the message window at the lower left the progress of the build. As expected, an error was encountered, so the build was not completed nor downloaded. Look above the final report indicating there was an error for the line (in red) indicating the specific error found. Note the report identifying what problem the compiler found. Now – double-click on the reported error and observe the file with the flaw is opened, and the cursor is set to the line in question. This technique should be used routinely to clean up flaws in newly authored or modified code, so keep it in mind as a valuable time saving trick.

3. Repair the typo in the variable name. While at this line, make another change: instead of testing for “1” instead test for “0”. Rebuild and retest. With the change in the code, how does switch 0 behave now?
D. (Optional) BIOS Instrumentation

One of the most powerful features of DSP/BIOS is the additional ability to perform temporal debugging. Once logical debug is completed, the ability to observe the load on the system and how events are progressing in real-time is a valuable system tuning aid. Later chapters in this workshop will provide all the details of the tools shown here, so for now it is not expected that you would have any ability to create their underlying components. Instead, the goals here are to observe some key BIOS instrumentation and consider how these can be helpful in system design. In addition, it is hoped that this preview will build enthusiasm for later chapters which will teach all aspects in the design, authoring, and use of these advanced temporal tools.

Access to the CPU Load Graph, Message Logs, Statistical Data, and an Execution Graph are all via CCS’s DSP/BIOS menu selection, as shown in the diagram below:

1. **Observe CPU Load:** The DSP/BIOS | CPU Load Graph window should already be in the bottom right corner of the CCS window (opened via Lab02.wks). A green trace in a black field represents the measured load on the CPU over time, with graduations noted on the left side. In the cells below, the current and peak loads are specified. Verify that switches 2 and 3 are in the ‘up’ position, and run the program again and note the CPU load, with the filter running and bypassed. Is the load what you’d expected?

2. **Rebuild in release mode:** Switch to release in the Active Configuration cell and re-specify the Project Build Options as per steps A5 and A6, above. What is the load now?

3. Another influence on load is the 2nd thread present in this project (the ‘dummy’ load). The amount of load is set by EVM DIP switches 2 and 3, as indicated in the diagram at the beginning of this lab. Note the loads imposed by various settings of these DIP switches. Also note that loads > 100% inhibit real-time instrumentation, and CPU load update ceases.)
4. **Message Logs:** Open the message log window via **DSP/BIOS | Message Log**. Float this window in the CCS workspace. Verify the Log Name specified is **logDipSw**. As before, toggle the switches that control the sw0 and sw1 values, noting the status of the DIP switch modifications in the **Message Log** window. The commands that are driving the messages are real-time versions of the common **printf** function. As a real-time function, note that no interruption to the performance (‘clicking sounds’) of the real-time system occurs when the host is sent data for display.

5. **View Execution Graph and Statistical Data:** Finally, to observe the execution graph and statistical data, load the workspace: **Lab02d.wks**. This workspace opened the two new panes and positioned the panes in the CCS window for a good layout.

6. **Execution Graph Management:** In the **Execution Graph** window, note the tskProcBuf and TSK0 threads. These represent the audio and dummy load functions, respectively. To make these easier to watch, right click on the Execution Graph, select **Property Page**, and uncheck all but tskProcBuf, tskLoadAndSwitch and Other Threads.

7. **Execution Graph Observations:** Change the load to setting 3, as indicated in the **Message Log** display. With a low load setting, tskLoadAndSwitch is seldom seen, but note now the increase in the running state (thick blue line) of the dummy load. Note also that TSK0’s process time was made so long that it exceeds several audio thread events. The execution graph is displaying an important BIOS feature – the ability of the BIOS scheduler to prioritize threads. Since the audio thread was made higher priority than the load, the load is preempted every time the audio thread needs to run. What do you think would happen if their priorities were reversed? In a later lab, this case will be tested, and you can determine if your answer was correct.

8. **Statistics Display Observations:** Observe the **Statistics View** window. A number of variables are displaying a range of statistical information: how many times they’ve been encountered in code, the total number of cycles they’ve consumed thus far, the maximum value of that item thus far, and the average value of all the times the value has been observed. Note again that all this data is being provided in real-time, without intrusion on the real-time threads the DSP is running.

9. **Statistics Display Management:** Right click on the **Statistics View** window and select clear to reset the counters.

10. **Statistics Display Setup:** Right-click on the **Statistics View** window and select **Property Page**... In the Units tab, click on STS Object tskProcBuf and select **Units:** Microseconds. Observe the change in the Statistics View window display, and note the average and maximum values of the execution time for this real-time thread. Since this TSK has priority over the load TSK, its execution time is relatively consistent.

Again, please note that all the tools previewed here will be fully covered in subsequent chapters and labs. Time permitting: feel free to experiment further with this lab and the tools and techniques seen here. Finally, review the CCS Reference Sheet that follows to determine if you are now familiar with all the items noted there. As noted earlier, you can remove that page and keep it nearby during subsequent labs if desired.
CCS Reference Sheet

Help: Contents, User manuals, Tutorial

--------------- Project Management

Project: New, Open, Save, Close, Add Files to Project, Build, Rebuild All,
Build Options:
  Compiler | Preprocessor: Include search path, Define Symbols
  Compiler | Basic: Target Version, Generate Debug Info,
  Speed v Size, Optimization Levels
  Linker | Basic: Output Filename, Map filename
  Linker | Basic: Library Search Path, Include Libraries

File: Workspace: Load Workspace, Save Workspace, Save Workspace As

Options | Customize:
  Debug Properties – Perform Go Main automatically
  Editor Properties – File Replacing Option: Automatically reload
  Program Load Options – Perform verification during Program Load,
    Load Program After Build,
    Clear All Breakpoints when Loading New Programs
  Control Window Display – Board name, Current loaded program,
    Current Project, Product Name
  Source file Names: Display full path
  Project close: Close all windows on Project Close
  Close projects: Close projects on exit Control Window

--------------- Debugging Techniques

File: Load Program, Reload Program

Debug: Run (F5), Halt (shift-F5), Step Over (F10), Run to Cursor (ctl-F10)
  Breakpoints: Enable All, Disable All, Delete All, Add (F9), Delete (F9)
  Animate (F12), Run Free,
  Reset, Restart, Go Main, Reset Emulator

View: Memory, Watch Window,

Edit: Memory, Register, Variable

--------------- Authoring Skills

File: New, Open, Close, Save, Save As, Save All,
  New – Source File (Ctl-N), DSP/BIOS Configuration

Edit: Cut (shift-delete), Copy (ctl-C), Paste (ctl-V), Select All (ctl-A)
  Delete (delete), Find/Replace (ctl-F), Undo (ctl-Z), Redo (ctl-Y)

BIOS API Help: highlight any BIOS API and press F1 for users guide reference

Options | Customize:
  Editor Properties – Tab stops, Auto-save Project and Files before build
Introduction

Hardware Interrupts or “HWI” are the most basic thread type managed by the DSP/BIOS scheduler. HWI are similar to conventional ISRs (interrupt service routines), except that BIOS permits the HWI to enjoy additional features and improved ease of use. HWI are present in almost all BIOS based systems, and are a likely mainstay of small project or those requiring low input-to-output latency.

Objectives

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

- Describe the concepts of foreground / background processing
- List details of the Idle thread
- Compare Hardware Interrupts (HWI) to ISR’s
- Demonstrate how to invoke Interrupt Preemption
- Describe the purpose of the Interrupt Monitor
- Create an HWI object using CCS Gconf tool
- Add an idle thread to a given CCS project
- Observe performance of threads using CCS

Module Topics

Hardware Interrupts - HWI

Concepts....................................................................................................................... 3-2
Idle (IDL) ........................................................................................................................................ 3-4
Hardware Interrupts (HWI)...................................................................................................... 3-7
Interrupt Preemption ........................................................................................................... 3-9
Interrupt Monitor .............................................................................................................. 3-12
Lab 3: An HWI-Based System.................................................................................................. 3-14
A. Project Creation ........................................................................................................... 3-16
B. TCF File Setup, HWI Definition, Project Build .......................................................... 3-17
C. Project Run ................................................................................................................ 3-18
D. Project Release Version ............................................................................................. 3-19
E. Adding an IDL Thread................................................................................................. 3-19
F. Save Completed Lab Exercise .................................................................................. 3-20
G. Optional Activities ................................................................................................... 3-20
Foreground / Background Scheduling

- IDL events run in sequence when no HWIs are posted
- HWI is ISR with automatic vector table generation
- Any HWI preempts IDL, HWI may preempt other HWI if desired
- If multiple HWI are present in IDL, control passes to highest priority HWI

Background scheduler allows you to defer less urgent processes from hardware interrupt service routines to the background.
Interrupt Enable Management Concepts

- Interrupt response is managed by a 2 level enable system:
  - Global Interrupt Enable (GIE) bit – indicates if any interrupts will be taken
  - Interrupt Enable (IE) register – indicates which interrupts are of interest
  - Pending interrupt signals are maintained in an Interrupt Flag (IF) register until responded to, and are automatically cleared when serviced

- On reset, GIE and all IE bits are cleared
- In main() whichever interrupts are desired initially should be enabled by ORing 1s to their corresponding bit position(s) in the IE
- When main() exits, GIE is automatically enabled as part of the start of the BIOS scheduler environment
- When an HWI is recognized: IF bit & GIE are cleared. GIE is cleared to avoid preemption amongst HWI. On return from the HWI, GIE status is restored
- Using the dispatcher on an HWI allows re-enable of GIE within the HWI if preemption is desired. Dispatcher also allows the selection of which other HWIs will be able to preempt the given HWI

Additional C64x interrupt management features:

- ICR for manually clearing a flag bit
- ISR for manually setting interrupts
- There are BIOS API for handling these, as well as setting the IER

C64x+ has an additional interrupt error event which can be used to indicate a missed interrupt

- If a flag is set, and another comes in, an error event is generated
- Can be mapped into a different (higher priority) interrupt or into an exception/NMI
Idle (IDL)

- **IDL**
  - Lowest priority - soft real-time - no deadline
  - Idle functions executes sequentially
  - Priority at which real-time analysis is passed to host

- **Likely IDL Activities**
  - Low power systems - idle the processor
  - Systems in test - instrumentation
  - User interfaces
  - Defragmentation
  - Garbage collection

Diagram:

- Various states and transitions:
  - Return from main() to Inactive
  - Preempted state
  - Started and Resume actions
  - Transition from Ready to Running
Creating a New Idle Object Via GCONF 1/4

Creating a new Idl Obj
1. right click on IDL mgr
2. select “Insert IDL”

Creating a New Idle Object Via GCONF 2/4

Creating a new Idl Obj
1. right click on IDL mgr
2. select “Insert IDL”
3. type object name
Creating a New Idle Object Via GCONF 3/4

Creating a new Idl Obj
1. right click on IDL mgr
2. select “Insert IDL”
3. type object name
4. right click on new IDL
5. select “Properties”

Creating a New Idle Object Via GCONF 4/4

Creating a new Idl Obj
1. right click on IDL mgr
2. select “Insert IDL”
3. type object name
4. right click on new IDL
5. select “Properties”
6. indicate desired
   • User Comment (FYI)
   • Function to run in IDL
   • Whether to include this function in the CPU load display
Hardware Interrupts (HWI)

**Hardware Interrupts**

- Much like “ISR”s (interrupt service routines)
- Vector table automatically rendered
- Add *interrupt* keyword in front of function declaration
- Context switch (save/restore of state of CPU around the HWI on the system stack) automatically performed when interrupt keyword is used
- Are a priority *foreground* activity that preempt background activity
- HWIs are taken in order of priority
- Default is one HWI does not preempt another: when a running HWI returns, then execution will pass to the highest priority HWI then available (or back to IDL if no HWI are flagged)

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Background</th>
</tr>
</thead>
</table>

**HWI Scheduling Example**

- Any HWI will preempt IDL
- Standard practice is that no HWIs preempt any other running HWI
- On return from an HWI, control is passed to highest pending HWI
- Is it always desirable to make high priority HWIs wait for lower priority HWIs to conclude before they are serviced?
Creating a New HWI Object Via GCONF 1/2

1. expand the HWI mgr
2. right click on desired HWI
3. select "Properties"

Creating a New HWI Object Via GCONF 2/2

1. expand the HWI mgr
2. right click on desired HWI
3. select "Properties"
4. select interrupt source and function to run
Interrupt Preemption

Adding Preemption to HWIs

- When preemption amongst HWIs is desired, default HWI scheduling can be manually overridden
- Developer can use the dispatcher in CCS to make any desired HWI preemptible
- Preemption can be on all higher numbered HWIs, or on any selected group of higher or lower HWI
- Adding the dispatcher increases context save and restore effort, some extra system overhead incurred
- Use of the dispatcher requires removing the interrupt keyword in the function declaration
- While seemingly desirable, HWI preemption will be seen to be only one of several scheduling options - handy in some cases, unneeded in others

Preemptive HWI Scheduling Example

Legend
- Running
- Ready

Any HWI will preempt IDL
HWI priority 3 does not preempt HWI priority 2
HWI priority 1 preempts HWI priority 2
note: if the dispatcher had been differently configured, HWI_a could have as easily preempted HWI_b, and HWI_c not so allowed
Enabling Preemption via the Dispatcher

To activate the dispatcher for a particular HWI:
- Right click on an HWI and select the "properties" option
- Select the Dispatcher tab in the properties dialog box
- Check the Use Dispatcher box
- Select HWIs that will preempt this HWI via the Interrupt mask
- Option: Arg field allows an argument to be passed to the HWI
- Be sure to remove ‘interrupt’ keyword in front of ISR when using dispatcher !!

Assembly Code Dispatch Option

- HWI_enter() and HWI_exit() are assembly macros
- Use _enter at the start of an HWI and _exit at the end of the HWI
- Allows BIOS calls to be correctly invoked between the functions
- User specifies preemption by other HWIs (IEMASK)
- GIE enabled on _enter and restored on _exit
- User specifies registers to save/restore (ABMASK, CMASK)
- User specifies cache control options (CCMASK)
- Cannot be used on HWIs that employ the BIOS dispatcher!
- Do not use interrupt keyword when using _enter and _exit !
- Usually use BIOS dispatcher, for final optimization consider _enter, _exit

```
myISR: HWI_enter ABMASK, CMASK, IEMASK, CCMASK
... SWI_post(&mySwi);
... HWI_exit ABMASK CMASK IERRESTOREMASK CCMASK
```
Comparison of Interrupt Options

- **Recommended: Use the BIOS dispatcher** as a first choice
  - Allows for selectable nesting of interrupts and BIOS scheduler calls
  - Easy to set up and manage via the config tool
- **Use HWI_enter and HWI_exit to optimize extremely speed critical HWI**
  - Can specify which registers to save, cache details, etc
  - Still allows BIOS calls and preemption
  - Requires knowing which registers to save for the given HWI
- **Interrupt keyword allows fast and small HWI – but no BIOS kernel API**
  - Any calls of BIOS API that prompt kernel scheduler action are prohibited
  - Nesting of HWI requires manual management of GIE and IER

<table>
<thead>
<tr>
<th></th>
<th>BIOS Dispatcher</th>
<th>Interrupt Keyword</th>
<th>HWI_enter, HWI_exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of use</td>
<td>Easy</td>
<td>Easy</td>
<td>Demanding</td>
</tr>
<tr>
<td>Post to scheduler?</td>
<td>Yes</td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>Chance of error</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Speed</td>
<td>Medium</td>
<td>Fast</td>
<td>Can be fastest</td>
</tr>
<tr>
<td>Code size</td>
<td>Smaller</td>
<td>Smaller</td>
<td>Larger</td>
</tr>
</tbody>
</table>

**ONLY CHOOSE ONE OF THE ABOVE OPTIONS PER HWI**

HWI API Summary

<table>
<thead>
<tr>
<th>HWI API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWI_enter</td>
<td>Tell BIOS an HWI is running, GIE enabled</td>
</tr>
<tr>
<td>HWI_exit</td>
<td>Tell BIOS HWI about to exit</td>
</tr>
<tr>
<td>HWI_enable</td>
<td>Turns on GIE bit, enables ISRs to run</td>
</tr>
<tr>
<td>HWI_disable</td>
<td>Sets GIE to 0, returns prior GIE state</td>
</tr>
<tr>
<td>HWI_restore</td>
<td>Restor GIE to given state before HWI_disable</td>
</tr>
<tr>
<td>HWI_dispatchPlug</td>
<td>Write a fetch packet into the vector table – dynamic ISR creation</td>
</tr>
</tbody>
</table>
Interrupt Monitor

**HWI Monitor Option**

**default configuration**

<table>
<thead>
<tr>
<th>Vector Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>br isr₀</td>
</tr>
<tr>
<td>br isr₁</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>br isrₙ</td>
</tr>
</tbody>
</table>

**monitoring isr₁**

<table>
<thead>
<tr>
<th>Vector Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>br isr₀</td>
</tr>
<tr>
<td>br stub₁</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>br isrₙ</td>
</tr>
</tbody>
</table>

- If monitor is other than “nothing”:
  - Stub function ‘plugged’ into vector table by DSP/BIOS
  - In turn, this will call your ISR

*Note: stubs are also used when the dispatcher is invoked*

---

**Setup of HWI Monitor Option**

To activate the monitor option for an HWI:

- Right click on an HWI; select “properties”
- Select the General tab in the dialog box
- Under “monitor” select parameter to observe
Setup of HWI Monitor Option 2/2

To activate the monitor option for an HWI:
- Right click on an HWI; select “properties”
- Select the General tab in the dialog box
- Under “monitor” select parameter to observe
- For ‘data value’, select its address / label
- Identify type (signed / unsigned) of datum
- Select STS function - covered later...

HWI Object

- HWI objects are pre-defined; you can assign properties with the configuration tool
- DSP/BIOS automatically sets up interrupt vectors
- Diagram is conceptual - not literal - object format
Lab 3: An HWI-Based System

In this lab, an audio filtering system will be implemented with a single HWI thread. Each time a single data value is received, the FIR filter is run and a single result will be output. This is the optimal topology for systems requiring minimum delay time (e.g.: control systems). It is also a style to consider for simple systems with few threads to manage. However, in more complex systems, HWI-only systems become less ideal as context switch overhead will be incurred every data event time. Other, better ways to construct these systems will be explored in subsequent labs.

For those not familiar with how a FIR filter is implemented, the diagram below depicts the key concepts of the solution as applied in this lab:
In this lab, you will:
- create a CCS project
- add the required source files to the project
- create a BIOS configuration (.TCF) file from a given EVM6437 seed file
- define the audio processing function as an HWI via GCONF
- set the required build options
- build, download, run, and debug the code on the EVM6437
- use a variety of CCS and BIOS debugging techniques to observe and interact with the code as it runs on the DSP

You will also have the opportunity to study the function encapsulated by the HWI to review how the underlying code was constructed. As you will see, the code within the HWI is the code to solve the given problem. The thread type presented to BIOS allows it to be independently managed by the BIOS scheduler. In general, neither of these aspects need be concerned about the other.

In this workshop, the labs will share a consistent style. All labs were designed to be developed in a single directory: `C:\BIOS\Labs\Work`. Most labs build on the solution from the prior lab. Solutions to each lab are found in `C:\BIOS\Sols\nn`, where `nn` is the lab number. The solution folder for a prior lab can be used as a starting point for a given lab, or as a means of reviewing (rather than implementing) a given lab, if time and interest so dictate.
A. Project Creation

Since the procedures for creating a project were already seen in the prior chapter, they are listed in a minimalized form here. If more detail is required in this section, refer back to the detailed instructions for these procedures in the prior lab, or ask the instructor for assistance.

1. Clear the contents of C:\BIOS\Labs\Work. Start CCS and connect to the EVM via Alt-C

2. Create a new project: Project | New
   - Project Name: myWork
   - Location: C:\BIOS\Labs\Work
   note: verify location directory before moving on!

3. Define the build options: Project | Build Options
   - Under Compiler, Basic, Target Version select: C64+ (mv6400+)
   - Under Compiler, Preprocessor, Include Search Path type: ..\HW; ..\Algos; $(BSL) and click OK

4. Add files to the project:
   - Copy audio-3.c from C:\BIOS\Labs\Algos to C:\BIOS\Labs\Work. Add the new copy to the project. Double-click on audio-3.c in the project folders on the left CCS pane to open it onto the workspace to the right. Note the constructs pointed out in the code listing below.
   - From C:\BIOS\Labs\Algos, add: fir.c and coeffs.c. The FIR algorithm is a simple sum-of-products implementation. For more information on coding and optimizing DSP algorithms, the C6000 Optimization Workshop may be of interest.
   - From C:\BIOS\Labs\HW, add: codec.c. This file manages the serial port and data converter and is not of primary interest in this workshop. The soon to be released Driver Workshop and online documentation is the venue for this area of study.
   - From C:\CCStudio_v3.3\boards\evmdm6437_v2\lib add the board support library evmdm6437bsl.lib to the project. Via this library a number of board specific operations can be performed, such as initializing the board, reading the DIP switches and controlling the LED lamps.

```c
interrupt void isrAudio(void) {
    static short i;
    // loop index
    static int dataIn, dataOut;
    // interface to MCBSP read/write
    static short dataOutL, dataOutR;
    // FIR results of L & R channels
    dataIn = MCBSP1_DRR_32BIT;
    // Get one stereo sample (L & R Data)
    buf[0] = (short)dataIn;
    // Place Left data sample in delay line
    buf[1] = (short)(dataIn >> 16);
    // Put Right data sample in delay line
    for (i = FIRSZ-2; i >= 0; i--)
        // for 2*(#coeffs-1)
        buf[i+2] = buf[i];
        // move all data down 1 pair
    if( sw0 == 1 ) {
        // If filtering is on...
        fir(&buf[0], &coeffs[sw1][0], &dataOutL, FIRSZ, 1);
        // left channel FIR
        FIR(&buf[1], &coeffs[sw1][0], &dataOutR, FIRSZ, 1);
        // right channel FIR
    }
    dataOut = 0x0000FFFF & dataOutL;
    // get left value for output
    dataOut |= 0xFFFF0000 & (dataOutR << 16);
    // or in right chan in MSBs
    else {
        // if filtering is 'off'
        dataOut = dataIn;
        // new input copied to output
        MCBSP1_DXR_32BIT = dataOut;
        // Send data to codec, (single channel)
    }
}
```
B. TCF File Setup, HWI Definition, Project Build

BIOS configuration (TCF) files define available target system hardware resources, how software sections are mapped to target memory, which BIOS objects to create in the system, and their configurations. Management of TCF files will be a significant part of this and succeeding labs.

1. Create a new TCF file: File | New | DSP/BIOS Configuration. Select the seed file ti.platforms.evmDM6437 and click OK. This file includes the hardware specifications of the EVM, such as the memory populated on the board.

2. To allow for dynamic allocations used in later labs, go to System | MEM – Memory Section Manager, right-click, select Properties, and uncheck No Dynamic Memory Heaps and click on the OK buttons. Assign some IRAM as a heap via System | Memory | IRAM, right-click, select Properties, and check create a heap in this memory and click OK. Finally, go back to System | MEM – Memory Section Manager | Properties and in the General tab specify IRAM in both Segment for... drop down windows.

3. Define an HWI to be managed by the BIOS scheduler:
   • Open the Scheduling folder and the HWI sub-folder
   • Right-click on HWI_INT4 and select Properties
     – interrupt selection number : 51
     – function: _isrAudio (the function in audio-3.c this HWI will call)
     – on the Dispatcher tab, check Use Dispatcher

4. Save the TCF file: File | Save As... as myWork.tcf in directory C:\BIOS\Labs\Work

5. Add myWork.tcf to the project. Note the automatic inclusion of two extra Generated Files.

6. In the Project View window, right click on myWork.tcf (in the DSP/BIOS Config folder) and select Compile File. Verify successful completion in the Output window.

7. Add the linker command file generated in the step above, myWorkcfg.cmd, to the project. In addition to the manually added CMD file, the inclusion of the TCF file automatically added the two Generated Files items, and two header files in the Include folder (visible once the project is built or you issue the command Project | Scan All File Dependencies).

8. Save the project: Project | Save

9. Verify that the Active Configuration display reads Debug (i.e.: not Release).

10. Build the project: Project | Build. A status window should open showing the files being created and indicating the success or problems encountered with each. If there are errors, try to determine and repair them, or call the instructor for assistance. Once the build is successfully completed, the code should automatically download. Verify that a yellow arrow, indicating the current PC (Program Counter) value, appears in audio-3.c at the beginning of the main function.
C. Project Run

1. Start an audio source playing on the PC, and verify the audio I/O cables are in place.

2. Run the project: F5, <alt>Debug | Run, or via the icon. Music should begin playing with normal (full range / no filter effect) sound quality.

3. Enable the filter function via a watch window variable:
   - Open a watch window via: View | Watch Window
   - In the newly opened watch window, click on the tab to “Watch 1”
   - Add sw0 to the watch list by selecting and dragging the variable from main.c to the watch window
   - Click on the value (currently 0) of sw0 - it should then be highlighted in a blue 'edit mode' color – and change the value to '1' and press 'enter'

   The audio should sound distinctly 'murky' as all the high frequencies are now being filtered out. Try changing the enable back off and on again to verify this control and its use.

4. Control the coefficient set in use: Add the sw1 variable to the watch window as above. Try changing the value between 0, 1, and 2 which should select low-, high-, and all-pass filters, respectively.

5. Try out the use of graphical controls available in CCS as defined in a “GEL” file:
   - Load control.gel (gel subdirectory of labs folder)
   - Go to the CCS "GEL" menu, look for the FilterControl sub list, select each GEL there
   - The "On" slider controls the sw0 variable - slide the control up and down and note the effect on the sound as well as the variable value in the watch window
   - The "Tone" slider allows the selection of Low-pass, High-pass, and All-pass filters. Note the effect of this slider on the sw1 watch window variable value and the audio sound

6. Arrange the windows on the screen to make all key information visible.

7. Save this debug session configuration via: File | Workspace | Save Workspace As ...
   Use any name you like and save in the working directory C:\BIOS\Labs\WKS

8. Try loading a different workspace via: File | Workspace | Load Workspace ... From the dialog box, select: Lab-3.wks from c:\BIOS\Labs\WKS. Feel free to use the new one or recall your prior workspace.

9. Measure the load this lab presents to the DSP: Open the DSP/BIOS | CPU Load Graph window (or via icon: ). What load do you observe over a range of sw0 and coefficient values? Note your observations here:

10. Halt the execution of the code via: <shift>F5, <alt>Debug | Halt, or the icon.
D. Project Release Version

Was this the range of CPU usage you would have predicted? Probably not - it seems quite high. Why was this so? The main factor is that code built in 'debug' configuration is not optimized. In this next section, building and measuring an optimized build of the code will be investigated.

1. Change the **Active Configuration** window from 'Debug' to **Release**

2. Define the release-version **Project | Build Options ...**
   
   Follow the instructions outlined in step A-3 above (don’t include the DEBUG item this time)

3. **Build** the Project. Wait for the code to download and the PC to arrive at **main**

4. **Run** the code. **Enable the filter** by using sw0 in the watch window and **examine the new CPU load.** How does this compare to what you observed in step C-9?

---

**Note:** This kind of performance improvement is not unusual. The use of optimization should be kept in mind as a key step in obtaining the maximum benefit of the DSP.

5. **Halt** the debugger.

E. Adding an IDL Thread

When the DSP is not processing the HWI, it loops through a list of idle functions specified in the TCF file. These functions are serviced in succession as time permits. The code that allows BIOS data to be uploaded to the host PC during debug resides in some of these IDL threads. Users can add any additional functions desired to the IDL thread. In this case, the ability to read the settings of DIP switches 1 and 2 on the EVM and have their settings control the sw0 and sw1 variables will be added. The code to implement the reading of the switches is already written. In this lab, the goal will be to integrate the function into the project as a new IDL object.

1. **Add** the file dipMonitor.c (in the HW directory) to the project.

2. **Open** the dipMonitor.c file and **review the code** briefly. Note the convenience of the BSL (Board Support Library) I2C functions and how they are used here to read the DIP switches of the EVM board.

3. **Create** an IDL object in the TCF file. **Right click** on the IDL Function Manager and select **Insert IDL.** Give the new IDL object a name of your choosing.

4. **Associate** the readDipSwitches() function with the new IDL object: **Right click** on the **new IDL object**, select **Properties** and enter _readDipSwitches in the function field.

---

**Hint:** It is important that the tag “readDipSwitches” is preceded by an underscore symbol to denote that it is a C-generated function as opposed to an assembly function. C functions are renamed with a preceding underscore by the C compiler, and failure to include this in the tag will cause an error at the linker stage of the projet build.
5. **Add** a call to the `initDipSwitches()` function to `main()`. This function simply initializes the variables used in the `readDipSwitches()` function. In general, BIOS based systems often place initialization functions in `main()`.

6. **Add** "dipMonitor.h" to the list of inclusions at the top of `audio-3.c`.

7. **Build, load,** and **run** the project.

8. **Verify the correct operation of switch 0 and switch 1**

   Switch 0 should toggle the sw0 and switch 1 select between the LPF and HPF. Note also that the GEL sliders and watch values also allow these control. The `readDipSwitches()` function was written to only have effect when the DIP switch is changed, thus allowing manual debug changes to also be effective.

9. **Save the completed project**  

10. **If desired, note again the CPU loads.**

    Have they changed significantly from prior observation?

11. **Halt the debugger.**

**F. Save Completed Lab Exercise**

Using windows explorer, **create a new folder** named 03 in `c:\BIOS\mySols` and **copy** all files from `C:\BIOS\Labs\Work` into this new folder.

**Note:** When re-running saved projects, **move** the saved directory back under the `C:\BIOS\Labs` directory, so as to keep the relative locations for the files in the Algos, HW and other directories consistent to their original state. Incremental work could have been saved in folders under the Labs directory directly, but the choice to place them under the `mySols` directory was made so as to reduce ‘clutter’ in the main Labs folder during the class.

**G. Optional Activities**

Time permitting, feel free to examine any code elements of interest, or to try out any code changes you like to `lab03.c`. Avoid changing `fir.c` or any of the HW directory files, as these will be needed in their current state in later labs, and are not really a relevant part of the study of BIOS anyway. **Suggestion:** you may wish to restore the solution just saved in directory 03 back into the work directory as a more reliable starting point for the next lab once you are through with any further experimentation.
Software Interrupts - SWI

Introduction

In this chapter the second BIOS thread type – the Software Interrupt, or “SWI” will be investigated. Comparisons and contrasts to the previously covered HWI will be made. A variety of options for posting SWIs will be considered including examples of when each type might be preferred.

Objectives

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

- Describe the basic concepts of SWIs
- Demonstrate how to post a SWI
- Describe the SWI object
- List several SWI posting options
- Define the benefit of each SWI posting method
- Add a SWI to an HWI-based system

Module Topics

Software Interrupts - SWI................................................................................................................................. 4-1

- Concepts.................................................................................................................................................. 4-2
- Posting a SWI........................................................................................................................................... 4-5
- The SWI Object....................................................................................................................................... 4-9
- SWI API Review.................................................................................................................................... 4-11
- Queues (QUE)......................................................................................................................................... 4-12
- Block FIR Concepts ............................................................................................................................... 4-14
- Lab 4: SWI-Based System .................................................................................................................... 4-19
  A. SWI Based procBuf()....................................................................................................................... 4-19
  B. Passing Buffers Via QUEues............................................................................................................. 4-20
  C. (Optional) Interrupt Keyword / Dispatcher Conflict......................................................................... 4-22
Concepts

New Paradigm: DSP/BIOS Scheduler

- **SWI_post** is equivalent to setting ISR flag
- **Scheduler** replaces the while loop
- **SWI manager** is like ‘if’ test with no overhead

Hardware and Software Interrupt System

Execution flow for flexible real-time systems:

- **HWI**
  - Fast response to interrupts
  - Minimal context switching
  - High priority for CPU
  - Limited number of HWI possible

- **SWI**
  - Latency in response time
  - Context switch performed
  - Selectable priority levels
  - Execution managed by scheduler

- **DSP/BIOS** provides for HWI and SWI management
- **DSP/BIOS** allows the HWI to post an SWI to the ready queue
DSP/BIOS Preemptive Scheduler

Hardware Interrupts (HWI)
- Urgent response time
- Often at “sample rate”
- Microseconds duty cycle
- Preemptive or non-preemptive

Software Interrupts (SWI)
- Flexible processing time
- Often at “frame rate”
- Milliseconds duty cycle
- Preemptive

Idle (IDL)
- Best Effort
- Sequential Execution

BIOS: Prioritized Scheduling

- HWI: Collect data into frame/buffer, perform minimum processing
- SWI: Process each datum in buffer
- IDL: Runs when no real-time events are active
- HWI preempt SWI - new data is not inhibited by processing of frame
**State Diagrams: IDL, HWI, SWI**

- **IDL**
  - Lowest priority - soft real-time - no deadline
  - Idle functions executes sequentially
  - Priority at which real-time analysis is passed to host

- **HWI & SWI**
  - Encapsulations of functions with priorities managed by DSP/BIOS kernel
  - Run to completion (cannot be suspended or terminated prior to completion)
  - Runs only once regardless of how many times posted prior to execution
Posting a SWI

Scheduling Rules

Highest Priority

- HWI
- SWI_b (p2)
- SWI_a (p1)

Lowest Priority

- IDL

Legend

- Running
- Ready

- \texttt{SWI\_post(&mySwi)}: Unconditionally post a software interrupt (in the ready state)
- If a higher priority thread becomes ready, the running thread is preempted
- SWI priorities from 1 to 14
- Automatic context switch (uses system stack)

- \texttt{SWI\_post(&SWI\_b)}

Processes of same priority are scheduled first-in first-out
Posting a SWI from an HWI

- Problem: Scheduler not aware of interrupt!
- If ISR posts a higher priority SWI, the scheduler will run that SWI in the context of the HWI - not usually desired

Using the Dispatcher with HWI

- Solution: Use the Dispatcher
- Some APIs that may affect scheduling: SWI_post, SWI_andn, SWI_dec, SWI_inc, SWI_or, SEM_post, PIP_alloc, PIP_free, PIP_get, PIP_put, PRD_tick
Scheduling Strategies

- Most important “Deadline Monotonic”
  - Assign higher priority to the most important process
- Rate monotonic analysis
  - Assign higher priority to higher frequency events
  - Events that execute at the highest rates are assigned highest priority
  - An easy way to assign priorities in a system!
  - Systems under 70% loaded guaranteed to run successfully (proofs for this in published papers)
  - Also allows you to determine scheduling bounds
- Dynamic priorities
  - Raise process priority as deadline approaches

DSP/BIOS: Priority-Based Scheduling

Diagram showing the scheduling of different interrupts and processes in a DSP/BIOS environment.
Another Scheduling Example

The BIOS Execution Graph provides this kind of information to assist in temporal debugging.
The SWI Object

Creation of SWI with Configuration Tool

1. right click on SWI mgr
2. select "Insert SWI"
3. type SWI name
4. right click on new SWI
5. select "Properties"
6. indicate desired
   • function
   • priority
   • mailbox value

SWI Attributes : Manage SWI Properties

- Allows programmer to inspect and modify key SWI object values
- Do not modify fields on preempted or ready to run SWI recommended: implement during lower priority thread
- Priority range is 1 to 14, inclusive
- Example - changing a SWI’s priority to 5:

```c
extern SWI_Obj swiProcBuf;
SWI_Attrs attrs;
SWI_getattrs (&swiProcBuf, &attrs);
attrs.priority = 5;
SWI_setattrs (&swiProcBuf, &attrs);
```
The SWI Object

SWI Structures (from swi.h and fxn.h)

- SWI_Attrs contains the most commonly used SWI object elements
- SWI_getattrs and SWI_setattrs allow well defined access to these elements

```c
typedef struct SWI_Attrs {
    SWI_Fxn fxn;
    Arg arg0;
    Arg arg1;
    Int priority;
    Uns mailbox;
} SWI_Attrs;
```

```c
typedef struct SWI_Obj {
    Int lock;
    Ptr ready;
    Uns mask;
    Ptr link;
    Uns initkey;
    Uns mailbox;
    FXN_Obj fxnobj;
    Int stlock;
    STS_Obj *sts;
} SWI_Obj;
```

- SWI object can be directly access also, if desired, as per these examples:
  ```c
  myValue = mySwi.fxnobj.arg1;
  mySwi.fxnobj.arg0 = 7;
  ```

Managing Thread Priorities via GCONF

- Drag-and-drop SWIs in list to vary priority
- Priorities range from 1-14

- Scheduler is invoked when SWI is posted
- When scheduler runs, control is passed to the highest priority thread
- Equal priority SWIs run in the order posted
SWI API Review

SWI Post and SWI Mailbox Overview

<table>
<thead>
<tr>
<th>API</th>
<th>Allows you to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_inc</td>
<td>Know how many times the SWI was posted before it ran</td>
</tr>
<tr>
<td>SWI_dec</td>
<td>Post N times before the SWI is scheduled – a countdown</td>
</tr>
<tr>
<td>SWI_or</td>
<td>Send a single value to the SWI when posting - signature</td>
</tr>
<tr>
<td>SWI_andn</td>
<td>Only post the SWI when multiple posters all have posted</td>
</tr>
</tbody>
</table>

- If the value of the mailbox is needed by the SWI, use SWI_getmbox() which returns the value of the mailbox when the SWI was posted.
- Note: this is a ‘shadow’ value for use within the SWI – BIOS manages a second mailbox for the next posting of the SWI
- After each posting, the mailbox is reset to the initial condition specified in the SWI object

SWI API Summary

<table>
<thead>
<tr>
<th>SWI API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_post</td>
<td>Post a software interrupt</td>
</tr>
<tr>
<td>SWI_andn</td>
<td>Clear bits from SWI's mailbox; post if becomes 0</td>
</tr>
<tr>
<td>SWI_or</td>
<td>Or mask with value contained in SWI's mailbox field</td>
</tr>
<tr>
<td>SWI_inc</td>
<td>Increment SWI's mailbox value</td>
</tr>
<tr>
<td>SWI_dec</td>
<td>Decrement SWI's mailbox value; post if becomes 0</td>
</tr>
<tr>
<td>SWI_getattrs</td>
<td>Copy SWI attribute from SWI object to a structure</td>
</tr>
<tr>
<td>SWI_setattrs</td>
<td>Update SWI object attributes from specified structure</td>
</tr>
<tr>
<td>SWI_getmbox</td>
<td>Obtain the value in the mailbox prior to SWI run</td>
</tr>
<tr>
<td>SWI_create</td>
<td>Create a SWI</td>
</tr>
<tr>
<td>SWI_delete</td>
<td>Delete a SWI</td>
</tr>
<tr>
<td>SWI_disable</td>
<td>Disable software interrupts</td>
</tr>
<tr>
<td>SWI_enable</td>
<td>Enable software interrupts</td>
</tr>
<tr>
<td>SWI_getpri</td>
<td>Return a SWI’s priority mask</td>
</tr>
<tr>
<td>SWI_raisepri</td>
<td>Raise a SWI’s priority</td>
</tr>
<tr>
<td>SWI_restorepri</td>
<td>Restore a SWI’s priority</td>
</tr>
<tr>
<td>SWI_self</td>
<td>Return current SWI’s object handle</td>
</tr>
</tbody>
</table>
Queues (QUE)

Queue Concepts

- QUE message is anything you like, starting with QUE_Elem
- QUE_Elem is a set of pointers that BIOS uses to manage a double linked list
- Items queued are NOT copied – only the QUE_Elem ptrs are managed!

```c
struct MyMessage {
    QUE_Elem elem;
    Int x[1000];
} Message1;
```

```c
typedef struct QUE_Elem {
    struct QUE_Elem *next;
    struct QUE_Elem *prev;
} QUE_Elem;
```

QUE_put(hQue,*msg3)  
add message to end of queue (writer)

*elem = QUE_get(hQue)  
get message from front of queue (reader)

How do you synchronize reader and writer?

Queue Usage

- QUE Properties
  - any number of messages can be passed
  - atomic API assure correct sequencing
  - no intrinsic semaphore

- Using QUE
  - Declare the QUE via the config tool
  - Define (typedef) the structure to queue – 1st element must be “QUE_Elem”
  - Fill the message(s) to QUE with the desired data
  - Send the data to the queue via QUE_put(&myQue, msg);
  - Acquire data from the queue via info=QUE_get(&myQue);

- Application Considerations
  - Two queues are needed to circulate messages between two threads

```c
typedef struct MsgObj {
    QUE_Elem elem;
    short *pInBuf;
    short *pOutBuf;
} MsgObj, *Msg;
```
### QUE API Summary

<table>
<thead>
<tr>
<th>QUE API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE_put</td>
<td>Add a message to end of queue – atomic write</td>
</tr>
<tr>
<td>QUE_get</td>
<td>Get message from front of queue – atomic read</td>
</tr>
<tr>
<td>QUE_enqueue</td>
<td>Non-atomic QUE_put</td>
</tr>
<tr>
<td>QUE_dequeue</td>
<td>Non-atomic QUE_get</td>
</tr>
<tr>
<td>QUE_head</td>
<td>Returns ptr to head of queue (no de-queue performed)</td>
</tr>
<tr>
<td>QUE_empty</td>
<td>Returns TRUE if queue has no messages</td>
</tr>
<tr>
<td>QUE_next</td>
<td>Returns next element in queue</td>
</tr>
<tr>
<td>QUE_prev</td>
<td>Returns previous element in queue</td>
</tr>
<tr>
<td>QUE_insert</td>
<td>Inserts element into queue in front of specified element</td>
</tr>
<tr>
<td>QUE_remove</td>
<td>Removes specified element from queue</td>
</tr>
<tr>
<td>QUE_new</td>
<td>....</td>
</tr>
<tr>
<td>QUE_create</td>
<td>Create a queue</td>
</tr>
<tr>
<td>QUE_delete</td>
<td>Delete a queue</td>
</tr>
</tbody>
</table>
Block FIR Concepts

Block FIR Filter Overview
- Read block of data to input buffer

Block FIR Filter Overview
- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer

A/D

SP

HWI

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2

3

4

5

95

96

97

98

99

0

1

0

0

0

1

2
Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer
- Repeat for BLOCKSIZE iterations

![Diagram of Block FIR Concepts]
Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer
- Repeat for BLOCKSIZE iterations
- Send output buffer to DAC
- Copy last N-1 samples to history pre-buffer

Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer
- Repeat for BLOCKSIZE iterations
- Send output buffer to DAC
- Copy last N-1 samples to history pre-buffer
- Repeat above steps...
**Double Buffer Management**

- Make data buffers size of block plus history
- Collect data in last ‘blocksize’ locations
- After buffer is processed, copy last ‘history’ values to top of other buffer

1. Get block of data
2. Start collecting next block while processing data
3. Prime for next block
4. Start collecting next block while processing data
5. Prime for next block
6. ...

**Interlaced Stereo Double Buffers**

```c
#define FIRSZ 64
#define HIST (2*FIRSZ-2)
short in[2][2*BUF+HIST];
short out[2][2*BUF];
for ( i = 0; i < HIST; i++ )
  pIn[i-HIST] = pPriorIn[i+2*BUF-HIST];
pPriorIn = pIn;

fir(pIn-HIST, &coeffs[cSet][0], pOut, FIRSZ, BUF);
fir(pIn+1-HIST, &coeffs[cSet][0], pOut+1, FIRSZ, BUF);
```

Note: the driver will be passed the address where new data is to be collected. Only the SWI will be aware of the history segment that precedes it.
Lab 4: SWI-Based System

This lab begins with a block based version of the audio filtering system. FIR processing is performed on a block (“buffer”) of data instead of being called with each new sample. Block processing is often desirable in multi-threaded systems, since context switching can be reduced to the block rate rather than the (often much) higher sample rate. This allows for more of the DSPs power to be applied to more complex and/or numerous activities.

In this lab, you will:

- begin with the solution from lab 3
- replace audio-3.c with isr.c, which does buffer IO, and proc.c, for the block FIR filtering
- modify the config file to make procBuf a SWI
- build and test the system and consider its range of usability
- modify isr.c and proc.c to circulate buffer ownership via queues (QUE)
- test this version and compare it to the prior version

All the files needed as a starting point for this lab procedure will be found in various directories under C:\BIOS\Labs\. If this particular chapter is of lesser interest, or if there is a time constraint, you may elect to skip over the authoring steps and use the solution files stored in directory C:\BIOS\Sols\04b, and proceed directly to seeing how the completed code looks and works.

Lab 4: Software Interrupts - SWI

- Create project using given SWI-based code
- Build, download, and test on the EVM
- Add QUEs to pass buffers between HWI and SWI
- Build, load, test, note differences
A. SWI Based procBuf()

In the steps below the HWI-only solution will be upgraded to one employing a SWI which implements block data FIR filtering.

1. **Open CCS** and the solution project from Lab 3. Verify the lab builds and runs properly.

2. **Remove audio-3.c** from the project and from the `Work` directory. It will be replaced by two files which separate the ISR and block data processing functions.

3. **Copy** from `C:\BIOS\Labs\Algos` to `C:\BIOS\Labs\Work`, `isr.c` and `proc.c`.

4. **Add isr.c** and `proc.c` to the project. Open `isr.c` and observe how the HWI collects a buffer of data from the serial port and passes it to the SWI once full. Open `proc.c`, and note how the delay line is managed and the offset of –HIST applied to the input buffers.

5. **Create a SWI**: In `myWork.tcf`, add a new SWI object named `swiProcBuf` which calls function `_procBuf`.

6. **Build, load, run**, and **test** the system. Note the CPU load in debug / release, filter on / off.

7. Open `proc.h` in `C:\BIOS\Labs\Algos`. Test how different buffer sizes affects the CPU load. Restore the original buffer size of 200 when done testing.

8. If desired, save the contents of the `Work` folder to a new folder, eg: `C:\BIOS\mySols\04a`.

---

```c
void isrAudio(void){
    static short bkCnt = 0;
    static short *pInBuf, *pOutBuf;
    static Bool N=0;
    if( bkCnt == 0 ) {
        pInBuf  = &in[N][HIST];
        pOutBuf = &out[N][0];
        dataIn = MCBSP1_DRR_32BIT; // get a stereo sample
        pInBuf[bkCnt] = (short)dataIn; // add L sample to L block
        pInBuf[bkCnt+1] = (short)(dataIn>>16); // R sample to R block
        if( bkCnt >= 2*BUF ) {
            pIn  = &in[N][HIST];
            pOut = &out[N][0];
            N ^=1;
            SWI_post(&swiProcBuf); // schedule SWI to process bufs
            bkCnt = 0;
        }
    }
    dataOut  = 0x0000FFFF & pOutBuf[bkCnt++]; // append R result
    MCBSP1_DXR_32BIT = dataOut; // send stereo value to DAC
    bkCnt+=2;
}

void procBuf() {
    for ( i = 0; i < HIST; i++ )
        pIn[HIST+i]=pPriority[i*2*BUF-HIST];
    pPriority = pIn;
    if( sw0 == 1 ) {
        for( i = 0; i < 2*BUF; i++ )
            pOut[i] = pIn[i];
    }
    else {
        for( i = 0; i < 2*BUF; i++ )
            pOut[i] = pIn[i];
    }
}
```

---

Lab 4a : procBuf() as a SWI

// for read/write to McBSP CSL
// monitors # samples collected
// pr to avail. in/out buf
// if there is no current buf
// if get in buf address
// get out buf address
// get a stereo sample
// add L sample to L block
// R sample to R block
// get L result
// append R result
// send stereo value to DAC
// inc.bk.ctr. by TWO samples
// get in buf address
// get out buf address
// schedule SWI to process bufs
// reset bk.ctr. for new buf's
B. Passing Buffers Via QUEues

1. **Create two queues**: In the config tool, under synchronization, add 2 QUE objects – qDev and qProc

2. **Create two messages for the queues**: In proc.h note the typedef of the message that will be passed by the queues. In proc.c, declare as globals, an array of two MsgObj's called bufPtrs

3. **Initialize the messages with the addresses of the two in and out buffers**: In the main() function, initialize the first bufPtr message with the addresses of input buffer 0, offset HIST and the base address of output buffer 0. *(Hint: the line of C code required to load an element of the message structure would be bufPtrs[0].pInBuf=&in[0][HIST]; ).* Repeat the process with the second message referencing buffer set 1. Note: the input buffers were offset by HIST so that the HWI would only fill the new data portion of the array, and not consider the preceding area reserved for procBuf to use for the requisite history buffer

4. **Prime the queue to the HWI with the two buffer pointer messages**: Once the messages have been initialized, place them into qDev for the HWI to acquire when it runs

5. **Add a pointer to receive messages from the queue**: In isr.c, create a static local variable intBufs of type Msg

6. **Use the queue message to determine which buffer set to use in the ISR**: Modify the if bkCnt=0 activity to begin by getting a message from the qDev queue. Then replace the loads of pInBuf and pOutBuf with the pointer information received from the queue. *(Hint: the line of C code required to load the pointer from the message pointer would be pIn = procBufs->pInBuf; )*

7. **Give the first set of buffers to the processing thread via the 2nd queue**: Modify the if bkCnt >= 2*BUF activity to replace the loads of the in and out pointers with a write to the qProc of the buffers now ready for processing prepared by the HWI

8. **Clean up prior references to the 'ping/pong' operator**: Remove all references to the variable N no longer required by the HWI

9. **Add a pointer to receive messages from the queue**: In procBuf, create a local variable procBufs of type Msg

10. **Use the queue message to determine which buffer set to use in the process**: Begin procBuf with a read of the message from qProc and initialize pIn and pOut with the information received there

11. **When finished, give back the buffer ownership**: At the end of the procBuf function, return the pointers back to the HWI via qDev

12. **Try out the new solution**: Build, download and run the code. Verify the performance

13. **Save** the contents of C:\BIOS\Labs\Work to C:\BIOS\mySols\Lab4b

14. **Optional**: time permitting, expand the solution to a 3 buffer system – something not readily possible with the pre-queue solution in part A of this lab
C. (Optional) Interrupt Keyword / Dispatcher Conflict

*Optional* (but valuable) : put back the interrupt keyword, rebuild and run again. What happens? Why? Remove the dispatcher check and try again. Note that in these configurations, the system fails, and it is very hard to figure out why using normal diagnostic efforts. What went wrong?

The use of the interrupt keyword allows the lowest overhead in launching an ISR, however this efficiency is had by *not* telling the BIOS scheduler about the HWI being run. This works fine as long as the HWI does not enable any other threads to run - such as we did with the `SWI_post()` operation. Since BIOS received a `SWI_post()`, and didn't know of the HWI running at that moment, it launches the SWI, and destabilizes the system, leading to the mysterious failure observed.

In the second case, where both the interrupt keyword and the dispatcher are used, a different problem occurs - two sets of context switches are invoked, but only one can be properly implemented, leaving an that is very hard to observe without painstakingly watching the system run line-by-line in assembly. All these headaches can be avoided, however, by following one of two simple disciplines. The first is foolproof: *never* use the interrupt keyword and always use the dispatcher. This method cost extra cycles of HWI context switch, but will always work as expected, and never requires further thought on the subject. The second method allows for the extra context cycles to be avoided when desired, but requires the user to be more cautious in authoring a system. Use interrupt keywords on HWIs that do not use BIOS APIs. On an HWI that uses a BIOS API, remove the interrupt keyword and enable the dispatcher for this HWI. Both methods work fine - as long as care is taken. Finally, if you are confronted with a system that fails mysteriously, remember to check for this potential problem early on in order to potentially save a lot of time. Another option to consider is to use only the dispatcher during development, and only switch to the use of the interrupt keyword during final system optimization, where appropriate per the above rules.
Introduction

In this chapter the concepts of authoring BIOS tasks (TSK) will be considered.

Objectives

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

- Describe the fundamental concepts of tasks
- Demonstrate the use of semaphores in tasks
- Author TSK code using streams to interface with IOM
- Create a TSK with the CCS GUI
- Describe the TSK object
- Explain the value of double buffers in DSP systems
- Modify single buffer code to use double buffers

Module Topics

Tasks - TSK ................................................................. 5-1

Comparison of Tasks to Software Interrupts ................................................................. 5-2
Task Scheduling ........................................................................................................... 5-3
Semaphores (SEM) .................................................................................................. 5-4
Task Object .............................................................................................................. 5-7
Review .................................................................................................................... 5-9
Lab 5: Adapting the SWI Based System to a TSK ...................................................... 5-10
Comparison of Tasks to Software Interrupts

Scheduler States: TSK vs SWI

<table>
<thead>
<tr>
<th>SWI</th>
<th>TSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created</td>
<td>Created</td>
</tr>
<tr>
<td>Inactive</td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>Released</td>
</tr>
<tr>
<td>Ready</td>
<td>Ready</td>
</tr>
<tr>
<td>Blocked</td>
<td>Blocked</td>
</tr>
<tr>
<td>Running</td>
<td>Running</td>
</tr>
<tr>
<td>Terminated</td>
<td>Terminated</td>
</tr>
</tbody>
</table>

Tasks are:
- ready to run when created
  - by BIOS startup if specified in GCONF
  - by TSK_create() in dynamic systems *(in later module)*
- preemptive
- blocked when pending on an unavailable resource
- returned to ready state when resource is posted
- may be terminated when no longer needed

SWI vs. TSK

<table>
<thead>
<tr>
<th>Feature</th>
<th>TSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preemptable</td>
<td>✓</td>
</tr>
<tr>
<td>Block, Suspend</td>
<td>✓</td>
</tr>
<tr>
<td>Delete prior to completion by other threads</td>
<td>✓</td>
</tr>
<tr>
<td>User Name, Error Number, Environment Pointer</td>
<td>✓</td>
</tr>
<tr>
<td>Can interface with SIO</td>
<td>✓</td>
</tr>
<tr>
<td>Context switch speed</td>
<td>slower</td>
</tr>
<tr>
<td>Context preserved across accesses to thread</td>
<td>✓</td>
</tr>
<tr>
<td>Can call SEM_pend()</td>
<td>✓</td>
</tr>
<tr>
<td>API callable by</td>
<td>C</td>
</tr>
</tbody>
</table>

* SEM_pend with timeout of 0 is allowed
# Task Scheduling

**DSP/BIOS Scheduler**

- **HWI (Hardware Interrupts)**
  - HWI priorities set by hardware
  - Fixed number, preemption optional
- **SWI (Software Interrupts)**
  - 14 SWI priority levels
  - Any number possible, all preemptive
- **TSK (Tasks)**
  - 15 TSK priority levels
  - Any number possible, all preemptive
- **IDL (Background)**
  - Continuous loop
  - Non-realtime in nature

- All TSKs are preempted by all SWIs and HWIs
- All SWIs are preempted by all HWIs
- HWIs preemption is under user control (inhibited by default)
- In absence of HWI, SWI, and TSK, IDL functions run in loop

**Thread Preemption Example**

Events over time

```
HWI                     SWI 2                     SWI 1
        post return     post return     post return
SWI 1      pend return     pend return     pend return
TSK 2      pend            pend            pend
           sem2           sem2           sem2
TSK 1      interrupt      interrupt      interrupt
           sem1           sem1           sem2
            return      post return     post return
           pend            pend
            return      post return     post return
               return      return
             pend            pend
                return      return
```
Semaphores (SEM)

Semaphores (SEM)

Task Code Topology - SEM Posting

Void taskFunction(...)
{
/* Prolog */
while ('condition'){
    SEM_pend()
/* Process */
}
/* Epilog */
}

- Initialization (runs once only)
- Processing loop - option: termination condition
- Wait for resources to be available
- Perform desired DSP work...
- Shutdown (runs once - at most)

- TSK can encompass three phases of activity
- SEM can be used to signal resource availability to TSK
- SEM_pend() blocks TSK until next buffer is available
Semaphore Pend

`SEM_pend(&sem, timeout)`

- **Pend**
  - `timeout = 0`
  - `Count > 0`

- **Decrement count**

- **Block task**
  - `timeout expires`

- **SEM posted**

- **Return FALSE**

- **Return TRUE**

Semaphore Structure:
- Non-negative 16-bit counter
- Pending queue (FIFO)

```
#define SYS_FOREVER (Uns) -1 // wait forever
#define SYS_POLL (Uns) 0    // don't wait
```

Semaphore Post

`SEM_post(&sem)`

- **Post**
  - Increment count
  - **False**
    - **Task pending on sem?**
      - **False**
      - **Return**
      - Task switch will occur if higher priority task is made ready
      - **True**
        - **Ready first waiting task**

Semaphore Structure:
- Non-negative count
- Pending queue (FIFO)
Semaphores (SEM)

Static Creation of SEM

Creating a new SEM Obj
1. right click on SEM mgr
2. select “Insert SEM”
3. type object name
4. right click on new SEM
5. select “Properties”
6. indicate desired
   • User Comment (FYI)
   • Initial SEM count

```javascript
var mySem = SEM.create("mySem");
mySem.comment = "my SEM";
mySem.count = 0;
```

SEM API Summary

<table>
<thead>
<tr>
<th>SEM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM_pend</td>
<td>Wait for the semaphore</td>
</tr>
<tr>
<td>SEM_post</td>
<td>Signal the semaphore</td>
</tr>
<tr>
<td>SEM_pendBinary</td>
<td>Wait for binary semaphore to = 1</td>
</tr>
<tr>
<td>SEM_postBinary</td>
<td>Write a 1 to the specified semaphore</td>
</tr>
<tr>
<td>SEM_count</td>
<td>Get the current semaphore count</td>
</tr>
<tr>
<td>SEM_reset</td>
<td>Reset SEM count to the argument-specified value</td>
</tr>
<tr>
<td>SEM_new</td>
<td>Puts specified count value in specified SEM</td>
</tr>
<tr>
<td>SEM_ipost</td>
<td>SEM_post in ISR – obsolete – use SEM_post</td>
</tr>
<tr>
<td>SEM_create</td>
<td>Create a semaphore</td>
</tr>
<tr>
<td>SEM_delete</td>
<td>Delete a semaphore</td>
</tr>
</tbody>
</table>
### Task Object Concepts...

**Task object:**
- Pointer to task function
- Priority: changeable
- Pointer to task's stack
  - Stores local variables
  - Nested function calls
  - Makes blocking possible
  - Interrupts run on the system stack
- Pointer to text name of TSK
- Environment: pointer to *user defined* structure:

```c
TSK_setenv(TSK_self(), &myEnv);
```

```c
hMyEnv = TSK_getenv(&myTsk);
```
typedef struct TSK_Obj {
    // from TSK.h
    KNL_Obj kobj;          // kernel object
    PTR stack;             // used w TSK_delete()
    size_t stacksize;     // ditto
    int stackseg;         // stack alloc'n RAM
    String name;          // printable name
    PTR environ;          // environment pointer
    int errno;            // TSK_seterr(),_geterr()
    bool exitflag;        // FALSE for server tasks
} TSK_Obj, *TSK_Handle;

typedef struct TSK_Config {
    int STACKSEG;         // task priority
    int PRIORITY;         // stack supplied
    size_t STACKSIZE;     // size of stack
    PTR environ;          // stack alloc'n seg
    int stackseg;         // environment pointer
    String name;          // printable name
    bool exitflag;        // server tasks = false
    bool initstackflag;   // server tasks = false
} TSK_Config;

typedef struct TSK_Stat {
    TSK_Attrs attrs;    // task attributes
    TSK_Mode mode;      // running, blocked...
    PTR sp;             // stack ptr
    size_t used;        // stack max
} TSK_Stat;

typedef struct TSK_Attrs {
    int priority;       // task priority
    PTR stack;          // stack supplied
    size_t stacksize;   // size of stack
    int stackseg;       // stack alloc'n seg
    PTR environ;        // environment pointer
    String name;        // printable name
    bool exitflag;      // server tasks = false
    bool initstackflag; // server tasks = false
} TSK_Attrs;

typedef struct KNL_Obj {
    // from KNL.h
    QUE_Elem ready;    // ready/sem queue
    QUE_Elem alarm;    // alarm queue elem
    QUE_Elem setpri;   // set priority queue
    QUE_Handle queue;  // task's ready queue
    int priority;      // task priority
    PTR sp;            // current stack ptr
    size_t timeout;    // timeout value
    int mode;          // blocked, ready, ...
    STS_Obj *sts;      // for TSK_deletatime()
    bool signalled;    // woken by sem or t-out
} TSK_Object

DSP/BIOS - Tasks - TSK

18
## TSK API Summary

<table>
<thead>
<tr>
<th>TSK API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSK_exit</td>
<td>Terminate execution of the current task</td>
</tr>
<tr>
<td>TSK_getenv</td>
<td>Get task environment</td>
</tr>
<tr>
<td>TSK_setenv</td>
<td>Set task environment</td>
</tr>
<tr>
<td>TSK_getname</td>
<td>Get task name</td>
</tr>
<tr>
<td>TSK_create</td>
<td>Create a task ready for execution</td>
</tr>
<tr>
<td>TSK_delete</td>
<td>Delete a task</td>
</tr>
</tbody>
</table>

*Most TSK API are used outside the TSK so other parts of the system can interact with or control the TSK*

*Most TSK API are to allow:*
- TSK scheduler management (Mod 7)
- TSK monitor & debug (Mod 8)
- Dynamic creation & deletion of TSK (Mod 10)

*TSK author usually has no need for any TSK API within the TSK code itself*
Lab 5: Adapting the SWI Based System to a TSK

The lab adaptation will include the following:
- In the TCF file, replace the processing SWI with a TSK and create a SEM
- In isr.c:
  - Post a SEM instead of the SWI
- In proc.c:
  - Add a while loop and SEM_pend to the procBuf() function
- Build, download, run, and verify the correct operation of the new solution
- Copy the solution files to C:\BIOS\mySols\05

For a more rigorous test of your skills – if time permits – you can attempt this lab given the information on this page alone. In most cases, it is recommended to follow the implementation steps on the next page. Note that given the experience gained in prior labs, the procedures on the next page are briefer and more demanding.

A solved set of these files are found in C:\BIOS\Sols\05. If this particular chapter is of lesser interest, or if there is a time constraint, you may instead copy the solution files to the Work directory, skip over the authoring steps, and move directly to seeing how the completed code looks and works.
Below are the steps required to adapt the SWI-based processing thread to a TSK-based version.

1. If necessary, start CCS and open the solution from lab 4b. Build the project and verify it performs properly

In myWork.tcf:

2. Replace the SWI that called procBuf with a TSK named tskProcBuf

3. Create a SEM named semBufRdy

In isr.c

4. Replace the SWI_post() with a SEM_post() of semBufRdy

In proc.c

5. Add a while(1) loop around all the iterative code in procBuf.

6. Add a SEM_pend on the newly created semaphore as the first line of the while loop

Having completed the adaptation steps you can now:

7. Build, load, run, and verify the correct operation of the new solution

8. (optional) Relocate the initialization of the messages and toDevQ to the prolog of the TSK. While not required, this makes the TSK a more complete (and instantiable) component

9. Using windows explorer, copy all files from C:\BIOS\Labs\Work to C:\BIOS\mySols\05
Streams - SIO

Introduction

In this chapter a technique to exchange buffers of data between input/output devices and processing threads will be considered. The BIOS ‘stream’ interface will be seen to provide a universal interace between I/O and processing threads, making coding easier and more easily reused.

Objectives

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

- Describe the concept of BIOS streams
- List the key stream API
- Adapt a Task to use stream interfacing
- Describe the benefits of multi-buffer streams
- Set up a stream via the configuration tool
- Describe how streams can interface to SWI

Module Topics

Streams - SIO................................................................................................................................. 6-1

Stream Concepts ......................................................................................................................... 6-2
Stream API ................................................................................................................................. 6-5
Adding Streams to Tasks .......................................................................................................... 6-9
Double Buffered Streams .......................................................................................................... 6-10
Adding Streams and Drivers to Projects .................................................................................. 6-12
Lab 6: Stream Based System .................................................................................................... 6-14
   A. Test the PSP Driver Project in the PSP Folder Tree .................................................... 6-14
   B. Move the PSP Driver Project to the “Work” Directory................................................ 6-15
   C. Adding the FIR Filter App to the Driver Example Code ............................................ 6-17
   D. Optional: “Clean” Initial Output Buffer Version ......................................................... 6-18
   E. Optional: Multi-Buffer Versions ................................................................................... 6-18
Stream Concepts

Why Not Use Direct Pointer Passing?

Problems with direct pointer passing:
- Adaptability: changes are within the threads
- Both driver and process need to be adapted
  - time consuming
  - ownership of both code sets?
  - opportunity to introduce errors

Preferred:
- Insulation between threads
  - Independent authorship
  - Increased and more direct reuse
  - Limits scope of required knowledge

Standardized IO Interface Benefits

Proposal: Standardized interface between IO and Process
- IO’s write to generic interface
- Process assumes the generic interface as data source/sink
- Each can be written independently of the other
- Maximized reuse – all I, P, O authors write to the same interface
- Major Bonus: Make interface a system integration control point
  Allows late-stage selection of interface properties
  - Size of buffers
  - Number of buffers
  - Which memories buffers reside in
- Key: Larger systems need to separate component authoring details from system integration adjustments
SIO Concepts

How is all this done?

- IO author will 'wrap' the basic HWI with code to i/f to the 'stream'
  (Drive details are in chapter 12)
- TSK controls and 'owns' the stream via SIO API
- TSK usually 'owns' the memory blocks passed between TSK and IOM
- TSK will 'issue' a block to the stream to begin data activity
  - Issue to an input device is usually of an 'empty' buffer to be filled
  - Issue to output device is usually 'full' buffer to be consumed
- TSK will 'reclaim' the issued buffer to:
  - read new input data or
  - obtain 'used' output buffer to write new data into
- Stream will block the TSK on reclaim if no buffer currently available
- Stream unblocks TSK when data buffer becomes available

SIO Tactical Benefits

- Stream API are part of BIOS and streams are managed by BIOS, so the user obtains this infrastructure with no coding effort on their part
- Stream 'decouples' the hard real-time nature of the IOM from the TSK
- Stream synchronizes TSK to IOM via SIO_reclaim being a blocking call
- Stream can maintain any desired number of buffers specified
- Data block size can be whatever is desired
- TSK and IOM can be written to obtain these parameters from the SIO
- Therefore – system integrators can use the specification of the stream properties to perform late-stage system tuning – adapting the number and size of buffers to optimize the performance of the TSK & IOM without changing any of the internal code of the TSK or IOM!
- SIO can be created dynamically, so this ‘tuning’ can even be performed as the DSP runs!
Stream Concepts

SIO Concepts

- Common I/O interface: between Tasks and Devices
  - Universal interface to I/O devices
  - Yields improved code maintenance and portability
  - Number of buffers and buffer size are user selectable
- **Unidirectional**: streams are input or output - not both
- Efficiency: uses pointer exchange instead of buffer copy
  - **SIO_issue** passes an "IOM Packet" buffer descriptor to Device via stream
  - **SIO_reclaim** waits for an IOM Packet to be returned by the DEV via stream
- Abstraction: TSK author insulated from device-specific functionality
  - BIOS manages two QUEues (todevice & fromdevice)
  - Driver author signals TSK buffer is ready via SEM
- Asynchronous: TSK and DEV activity is independent, synch’d by buffer passes
- Buffers: Data buffers must be created - by config tool or TSK

**IOM Packet**

prev(prev) QUE_ELEM next
Ptr addr
Uns size
Arg misc
Arg arg
Uns cmd
Int status

**MY_DSP_algo**

Taskissue reclaim
SIO reclaim
IOM Input
IOM Packet
output

reclaim

Output
IOM

---

6 - 4 DSP/BIOS - Streams - SIO
Stream API

Buffer Passing: `SIO_issue()` & `SIO_reclaim()`

```c
status = SIO_issue(hStream, pBuf, uSize, arg);
uSize = SIO_reclaim(hStream, &pBuf, pArg);
```

- `SIO_issue()` and `SIO_reclaim()` are the key stream API for exchanging buffers with IOM via the stream interface.
- `SIO_issue()` places a buffer in the stream.
- `SIO_reclaim()` requests a buffer from the stream. Note:
  - if no buffers are already available in the stream, this call can block until a buffer is made available from the IOM
  - *Never* attempt a reclaim prior to issuing buffer(s) into the stream first
  - *pBuf is where SIO_reclaim writes the address of the buffer returned

<table>
<thead>
<tr>
<th>type</th>
<th>param</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>status</td>
<td>report of success/failure of operation</td>
</tr>
<tr>
<td>Int</td>
<td>uSize</td>
<td># of used values in block - in nmadus (error = -)</td>
</tr>
<tr>
<td>SIO_Handle</td>
<td>hStream</td>
<td>handle to stream obj</td>
</tr>
<tr>
<td>Ptr</td>
<td>pBuf</td>
<td>pointer to buffer being stream transferred</td>
</tr>
<tr>
<td>Arg</td>
<td>arg</td>
<td>uncommitted gen’l purpose user argument</td>
</tr>
</tbody>
</table>

Obtaining Buffers: `SIO_staticbuf()`

```c
uSize = SIO_staticbuf(hStream, &pBuf);
```

- `SIO_staticbuf()` returns buffers for statically created streams where the ‘Allocate Static Buffers’ box was checked
- `SIO_staticbuf()` can only be called prior to any SIO_issue or _reclaim call

<table>
<thead>
<tr>
<th>type</th>
<th>param</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>uSize</td>
<td>size of block obtained - in nmadus (o if no more)</td>
</tr>
<tr>
<td>SIO_Handle</td>
<td>hStream</td>
<td>handle to statically created parent stream obj</td>
</tr>
<tr>
<td>Ptr</td>
<td>pBuf</td>
<td>pointer to buffer being stream transferred</td>
</tr>
</tbody>
</table>
Buffer Declaration/Creation Options

- A variety of methods are available for obtaining buffers for use in streams
- The table below compares the most common methods by consideration

<table>
<thead>
<tr>
<th>Setup Speed</th>
<th>Globals, Statics TSK Locals</th>
<th>SIO_staticbuf()</th>
<th>Dynamic Allocation malloc(), MEM_alloc(), etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM Re-use</td>
<td>Not possible without specific planning</td>
<td>Not possible without specific planning</td>
<td>RAM is easily returned to system for reuse</td>
</tr>
<tr>
<td>SIO Property Tracking</td>
<td>Manual – buffer properties not derived from SIO properties</td>
<td>Buffer properties are specified by SIO properties</td>
<td>Buffer properties can track stream properties via SIO_create arguments</td>
</tr>
<tr>
<td>Ease of routing an SIO’s buffers to a particular HW RAM</td>
<td>Requires use of pragma statements and linker routing</td>
<td>Each SIO’s buffers can come from any available HW RAM</td>
<td>Each SIO’s buffers can come from any available HW RAM containing a heap section</td>
</tr>
</tbody>
</table>

- Each system, and even individual streams might use a different approach, based on which of the noted criteria is of greatest importance in that case
- Ease of use? Not really a factor - none are significantly difficult to manage
  - All discussion to this point has been exclusively of static systems.
  - Dynamic Allocation techniques will be covered in module 10

Halting a Stream: `SIO_idle()`, `SIO_flush()`

```c
status = SIO_idle(hStream);
status = SIO_flush(hStream);
```

- `SIO_flush()` indicates to immediately return all buffers, regardless of state of data (sent, waiting, in progress)
- `SIO_idle()` is
  - identical to `_flush` for input streams,
  - directs output streams to complete sending any remaining ‘live’ data prior to return
  - blocks the calling task until all output data is sent
- In both cases, the underlying driver is idled prior to their return

<table>
<thead>
<tr>
<th>type</th>
<th>param</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Status</td>
<td>report of success/failure of operation</td>
</tr>
<tr>
<td>SIO_Handle</td>
<td>hStream</td>
<td>handle to stream obj</td>
</tr>
</tbody>
</table>
Managing the Underlying IOM: `SIO_ctrl()`

```c
status = SIO_ctrl(hStream, uCmd, arg);
```

**Example:**

```
SIO_ctrl(hMySio, DAC_RATE, 12000);
```

- `SIO_ctrl()` allows control and communication with the underlying IOM
- Not a frequently used SIO API
- Abilities offered with `SIO_ctrl()` are entirely the option of the IOM author
- Are by definition hardware-specific and unlikely matched with another IOM
- If IOM management only required when being bound to stream, can use parameter structure instead (see IOM chapter)
- `SIO_ctrl()` is ideal if parameter changes required after stream creation

<table>
<thead>
<tr>
<th>type</th>
<th>param</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>status</td>
<td>whatever return value the IOM author provided</td>
</tr>
<tr>
<td>SIO_Handle</td>
<td>hStream</td>
<td>handle to statically created parent stream obj</td>
</tr>
<tr>
<td>Uns</td>
<td>uCmd</td>
<td>command value passed to IOM – IOM specific</td>
</tr>
<tr>
<td>Arg</td>
<td>arg</td>
<td>uncommitted gen'l purpose user argument</td>
</tr>
</tbody>
</table>

“First Available” Stream: `SIO_select()`

```c
uMask = SIO_select(ahSioTab, nSio, uTimeout);
```

- Wait until one or more streams are ready for I/O
- `streamtab` defines which streams to pend on
- Useful for slow-device I/O
- Daemon task to route data from several sources
- **recommended: SCOM instead of SIO if possible - easier**

```c
SIO_Handle hSioIn, hSioOut, ahSioTab[2];
Uns   uMask;
ahSioTab [0] = hSioIn;
ahSioTab [1] = hSioOut;
uMask = SIO_select(ahSioTab, 2, SYS_FOREVER)
if (uMask == 0x1) {
    // service streamIn
}
if (uMask == 0x2) {
    // service streamOut
}
```

<table>
<thead>
<tr>
<th>type</th>
<th>param</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uns</td>
<td>uMask</td>
<td>ready stream ID</td>
</tr>
<tr>
<td>SIO_Handle</td>
<td>ahSioTab[]</td>
<td>table of streams</td>
</tr>
<tr>
<td>Int</td>
<td>nSio</td>
<td># of streams in table</td>
</tr>
<tr>
<td>Uns</td>
<td>uTimeout</td>
<td>max blocking time</td>
</tr>
</tbody>
</table>
# SIO API Summary

## Buffer Passing
- **SIO_issue**: Send a buffer to a stream
- **SIO_reclaim**: Request a buffer back from a stream
- **SIO_ready**: Test to see if stream has buffer available for reclaim
- **SIO_select**: Wait for any of a specified group of streams to be ready

## Stream Management
- **SIO_staticbuf**: Obtain pointer to statically created buffer
- **SIO_flush**: Idle a stream by flushing buffers
- **SIO_idle**: Idle a stream
- **SIO_ctrl**: Perform a device-dependent control operation

## Stream Properties Interrogation
- **SIO_bufsize**: Returns size of the buffers specified in stream object
- **SIO_nbufs**: Returns number of buffers specified in stream object
- **SIO_segid**: Memory segment used by a stream as per stream object

## Dynamic Stream Management (mod.10)
- **SIO_create**: Dynamically create a stream (malloc fxn)
- **SIO_delete**: Delete a dynamically created stream (free fxn)

## Archaic Stream API
- **SIO_get**: Get buffer from stream
- **SIO_put**: Put buffer to a stream
Adding Streams to Tasks

### Basic Task Code Topology

```c
Void taskFunction(…)
{
  /* Prolog */
  while ('condition'){
    SIO_reclaim();
    /* Process */
    SIO_issue()
  }
  /* Epilog */
}
```

- **Initialization** (runs once only)
- **Processing loop** - option: termination condition
- **Wait for resources to be available**
- **Perform desired DSP work...**
- **Shutdown** (runs once - at most)

---

### Task Coding Example

```c
void myTask()
{
  short inBuf[BUFSIZE];
  short outBuf [BUFSIZE];
  SIO_issue(&inStream, inBuf, sizeof(inBuf), NULL);
  SIO_issue(&outStream, outBuf,sizeof(outBuf), NULL);

  while (continue == TRUE) {
    inSize = SIO_reclaim(&inStream, (Ptr *)&inBuf, NULL);
    outSize = SIO_reclaim(&outStream, (Ptr *)&outBuf, NULL);
    for (i = 0; i < (outSize / sizeof(short)); i++) {
      outbuf[i] = 2*inbuf[i];
    }
    SIO_issue(&outStream, outBuf, outSize, NULL);
    SIO_issue(&inStream, inBuf, inSize, NULL);
  }

  SIO_idle(&inStream);
  SIO_idle(&outStream);
  SIO_reclaim(&inStream, (Ptr *)&inBuf, NULL);
  SIO_reclaim(&outStream, (Ptr *)&outBuf, NULL);
}
```

- **make stream buffers**
- **prime streams w buffers**
- **run loop w exit condition**
- **get buffers**
- **do DSP...**
- **send buffers**
- **turn off streams & IOMs**
- **get back buffers**

*Note: conceptual example only – cannot use static declaration ‘inBuf’ (and outBuf) as rtn ptrs...*
Double Buffered Streams

Double Buffer Concepts

- To maintain real-time throughput, it is often beneficial to provide two buffers to each stream:
  - One should always be available to the IOM
  - The other is available to the TSK for processing
    - Once processed by the TSK, it is re-issued back into the stream
    - Stream maintains buffers in a (BIOS) QUE until needed by IOM or TSK
    - After returning (issuing) the used buffer back into the stream, the TSK usually requests (reclaim) a new buffer from the stream to continue processing
      - If a block is available, the stream provides it to the TSK
      - Otherwise, TSK is blocked until IOM returns a buffer to the stream; then TSK is unblocked and given the buffer requested and now available
  - Allows for effective ‘concurrency’ (processing one buffer while filling another)
  - For real-time to be met, DSP processing should complete before the I/O does

Double Buffer Stream TSK Coding Example

```
// prolog – prime the process...
status = SIO_issue(&sioIn, pBufIn1, SIZE, NULL);
status = SIO_issue(&sioIn, pBufIn2, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
// DSP... to pBufOut1
status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
status = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
status = SIO_issue(&sioOut, pBufOut1, SIZE, NULL);
status = SIO_issue(&sioOut, pBufOut2, SIZE, NULL);
// while loop – iterate the process...
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pBufOutX, NULL);
    // DSP... to pBufOut
    status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
    status = SIO_issue(&sioOut, pBufOutX, SIZE, NULL);
}
// epilog – wind down the process...
status = SIO_flush(&sioIn);
status = SIO_idle(&sioOut);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn1, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn2, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut1, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut2, NULL);
```
**Triple Buffer Stream Coding Example**

```
//prolog – prime the process...
status = SIO_issue(&sioIn, pBufIn1, SIZE, NULL);
status = SIO_issue(&sioIn, pBufIn2, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn3, NULL);
// DSP... to pBufOut1
status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
status = SIO_issue(&sioOut, pBufOut1, SIZE, NULL);
status = SIO_issue(&sioOut, pBufOut2, SIZE, NULL);
status = SIO_issue(&sioOut, pBufOut3, SIZE, NULL);

//while loop – iterate the process...
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pBufOutX, NULL);
    // DSP... to pBufOut
    status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
    status = SIO_issue(&sioOut, pBufOutX, SIZE, NULL);
}

//epilog – wind down...
status = SIO_flush(&sioIn);
status = SIO_idle(&sioOut);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn1, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn2, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn3, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut1, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut2, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut3, NULL);
```

**“N” Buffer Stream Coding Example**

```
//prolog – prime the process...
for (n=0; n<SIO_nbufs(&sioIn); n++)
    status = SIO_issue(&sioIn, pBufIn[n], SIZE, NULL);

for (n=0; n<SIO_nbufs(&sioOut); n++)
    size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
    // DSP... to pBufOut[n]
    status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
}

for (n=0; n<SIO_nbufs(&sioOut); n++)
    status = SIO_issue(&sioOut, pBufOut[n], SIZE, NULL);

//while loop – iterate the process...
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pBufInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pBufOutX, NULL);
    // DSP... to pBufOut
    status = SIO_issue(&sioIn, pBufInX, SIZE, NULL);
    status = SIO_issue(&sioOut, pBufOutX, SIZE, NULL);
}

//epilog – wind down...
status = SIO_flush(&sioIn);
status = SIO_idle(&sioOut);
for (n=0; n<SIO_nbufs(&sioIn); n++)
    size = SIO_reclaim(&sioIn, (Ptr *)&pBufIn[n], NULL);
for (n=0; n<SIO_nbufs(&sioOut); n++)
    size = SIO_reclaim(&sioOut, (Ptr *)&pBufOut[n], NULL);
```
Since a stream connects a task to a driver, the first step in defining a stream is to register the driver (via the config tool). Modern BIOS drivers are a two-part solution – a “mini-driver” (MD) and a driver interface (here, “DIO”). Details of drivers are found in module 12.

### 1. Register the I/O Mini-Driver

#### DEV Table

<table>
<thead>
<tr>
<th>name</th>
<th>fxns</th>
<th>devid</th>
<th>params</th>
<th>type</th>
<th>devp</th>
</tr>
</thead>
<tbody>
<tr>
<td>uDevMyCodec</td>
<td>DSK6416_EDMA...FXN</td>
<td>0x0</td>
<td>0x0</td>
<td>IOM_Fxns</td>
<td>0x0</td>
</tr>
</tbody>
</table>

Include the files/library containing the IOM in the project.
Once the driver is defined, a stream can be created and bound to the driver. Number and size of buffers are up to the user. The selection of issue/reclaim model allows for a smaller SIO module to be built, since the archaic ‘get/put’ model code can be left out.
Lab 6: Stream Based System

PSP was created to reduce the effort required to develop systems based on TI DSPs. Drivers are often a time consuming and complicated part of software development, so having a library of these pre-made to draw upon can be very helpful. In addition to containing a large number of drivers, PSP also includes example projects which are a fast and convenient way to get started using those drivers.

Note: Since the directory path is rather deep, for readability, in the procedure below, the directory path: C:\dvsdk_1_01_00_15\psp_1_00_02_00\pspdrivers\ will be shown as $(PSP)$, which is identical a coding shorthand that will be seen in macro.ini later in this lab.

---

Lab 6: SIO Interfaced Task Thread

---

Lab 6: SIO Interfaced Task Thread

A – Test driver example in PSP tree
B – Move driver example into Work folder
C – Restore FIR application
   - Replace SEM & QUE, with SIO; delete isrAudio(), etc...
D&E – Optional: implement enhanced coding options

---

A. Test the PSP Driver Project in the PSP Folder Tree

1. Open CCS. Close any projects that may currently be open

2. Open project dm6437_evm_audio_st_sample.pjt in folder: $(PSP)\system\dm6437\bios\dm6437_evm\src\audio\sample\build
   (tip: open file C:\BIOS\Labs\Lab-6.txt which is an abbreviated form of these lab steps.
   Copy and paste the path into Explorer, then drag the pjt file from Explorer onto the CCS Project Folder)

3. Build the project and verify that the audio pass through example code is working properly

4. The example project employed a single DIO for both streams which is poor practice. Create two new DIO in the config tool called dioIn and dioOut. Set the device name property on both to udevCodec0

5. Open audio_sample.c and modify the 2 SIO_create() calls to use the new DIOs

6. Rebuild the project and verify performance is maintained
B. Move the PSP Driver Project to the “Work” Directory

While a fast way to get started using drivers, PSP example projects are located deep within the PSP folder tree, and while fine for validating initial operability, may not be the desired location for building up a more extensive project. As such, it is often desirable to relocate the example project to a new directory, and then build up the rest of the project on top of the base driver project. The steps below will show how to relocate the audio driver, and serve as a template for how to do so for other driver projects.

Reminder: Be sure to open file C:\BIOS\Labs\Lab-6.txt which has an abbreviated form of the lab steps, from which you can copy and paste the long file and path names, and therefore avoid typing errors and more quickly complete this mechanical procedure.

File Management

Several files will be copied from the PSP tree. Then these copies can be changed without altering the original driver files:

1. From $(PSP)\system\dm6437\bios\dm6437_evm\src\audio\sample
   copy audio_sample.c and psp_bios_audio_st_sample_main.c to C:\BIOS\Labs\Work

2. From $(PSP)\system\dm6437\bios\dm6437_evm\src\audio\src
   copy psp_audioCfg.c to the Work directory

3. From $(PSP)\system\dm6437\bios\dm6437_evm\src\audio\sample\build
   copy dm6437_evm_audio_st_sample.tcf to the Work directory

Project Setup

1. Open CCS

2. Starting with the solution file from the prior lab5 will allow useful build options from prior labs to continue to be available. Open the Lab5 solution project: myWork.pjt. Verify the project builds and runs properly

3. Since the initial goal is to reproduce the echo (audio pass-thru) application from the PSP folders, remove from the project (for now) all the .C source files

4. To follow prior lab style, delete the current myWork.tcf and rename
dm6437_evm_audio_st_sample.tcf to myWork.tcf

5. To implement the echo example, add to the project from the Work folder the following files:
   - audio_sample.c
   - psp_bios_audio_st_sample_main.c
   - psp_audioCfg.c

6. Also add from $(PSP)\drivers\i2c\sample:
   - i2cParams_evmdm6437.c
Project Configuration

The Lab 5 project file needs to be augmented with various library and include path statements from the driver example project file. The six steps below detail what needs to be done:

1. Rather than enter modifications to the project via menus and dialog boxes, it is possible to directly edit the file, which will speed the work here. Open myWork.pjt (right click on myWork.pjt in the project view window) and select Open for Editing.

2. Typing substitutions to shorten long path names are supported in CCS. Open macro.ini in C:\CCStudio_v3.3\cc\bin and review the substitutions that will be employed here.

3. To add the include paths needed by the driver, under ['Compiler" Settings: "Debug"] and ['Compiler" Settings: "Release"], before -i"..\HW" add:
   -i"$(EDMA)\inc" -i"$(PSP)\inc" -i"$(PSP_6437)"

4. To locate all the builder paths used, replace the DspBiosBuilder section with:
   -i"$(BIOS_Common)" -i"$(PSP_Dbg)" -i"$(EDMA_Dbg)" -l"palos_bios.lib" -l"mcasp_bios_drv.lib" -l"mcbsp_bios_drv.lib" -l"audio_bios_drv.lib" -l"edma3_drv_bios.lib" -l"edma3_rm_bios.lib" -l"edma3_drv_sample.lib" -l"log8.a64P"

5. To locate driver libraries for debug builds, add to ['"Linker" Settings: "Debug"]:
   -l"$(BIOS_Common)" -l"$(PSP_Dbg)" -l"$(EDMA_Dbg)" -l"palos_bios.lib" -l"mcasp_bios_drv.lib" -l"mcbsp_bios_drv.lib" -l"audio_bios_drv.lib" -l"edma3_drv_bios.lib" -l"edma3_rm_bios.lib" -l"edma3_drv_sample.lib" -l"log8.a64P"

6. To locate driver libraries for release builds, add to ['"Linker" Settings: "Release"]:
   -l"$(BIOS_Common)" -l"$(PSP_Rel)" -l"$(EDMA_Rel)" -l"palos_bios.lib" -l"mcasp_bios_drv.lib" -l"mcbsp_bios_drv.lib" -l"audio_bios_drv.lib" -l"edma3_drv_bios.lib" -l"edma3_rm_bios.lib" -l"edma3_drv_sample.lib" -l"log8.a64P"

7. In order for the above changes to be applied, save, close, and reload myWork.pjt.

File Modifications

1. The example project was configured for voice quality. To obtain audio quality conversions, Open psp_audioCfg.c and change the Sampling Rate to 44100 and Input Gain to 20.

2. To update the linker command file, in the Project Explorer window, right click on myWork.tcf and select Project | Compile File.

3. Now that all the adaptation is completed, build the project and test it in both the debug and release versions.

The 19 steps here demonstrate a simple mechanical process for how most PSP driver projects can be relocated, and may serve as a pattern to follow when using other PSP drivers in future projects.
C. Adding the FIR Filter App to the Driver Example Code

1. **Restore** to the project the following files:
   - from \Algos : fir.c
   - coeffs.c
   - from \HW : dipMonitor.c
   - from \Work : proc.c

   Open **proc.c** and make the following changes:

2. **Copy** from audio_sample.c to proc.c *(or proc.h)*:
   - from the ‘includes’ the includes of SIO and PSP_AUDIO
   - from the ‘externs’ the extern reference to the edma3init() function

3. In **main()** add a call to edma3init();

4. Add four local variables to procBuf : sioIn and sioOut of type SIO_Handle and short pointers pIn and pOut

5. Generally, streams would be declared in the config tool. However, the driver used here was designed to only be called dynamically. Therefore, we need to **begin** the procBuf function with two
   
   \[ \text{rtn} = \text{SIO\_create(handle, mode, size, attrs)}; \]

   calls - one for each stream. As the API is typed in, a prompt will appear with the prototype of the function args to help complete the call. Use the following table for the remaining values to complete the two lines:

<table>
<thead>
<tr>
<th>return value</th>
<th>sioIn</th>
<th>sioOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>“/dioIn”</td>
<td>“/dioOut”</td>
</tr>
<tr>
<td>mode</td>
<td>SIO_INPUT</td>
<td>SIO_OUTPUT</td>
</tr>
<tr>
<td>size</td>
<td>4*BUF</td>
<td>4*BUF</td>
</tr>
<tr>
<td>attrs</td>
<td>attrs</td>
<td>attrs</td>
</tr>
</tbody>
</table>

6. For **attrs**, a **local variable** attrs must be created of type SIO\_Attrs, initialized to SIO\_ATTRS. Prior to the SIO\_create calls, **modify** the model attribute to SIO\_ISSUERECLAIM, which will allow a considerably smaller stream object to be used.

7. **Copy** to the end of proc.c the I2C\_init() function from psp\_bios\_audio\_st\_sample\_main.c

8. Having moved the necessary elements from the demo files, audio_sample.c and psp\_bios\_audio\_st\_sample\_main.c can now be **removed from the project** and \Work

9. Since the PSP driver will be implementing the IO functionality, all the earlier driver components from prior labs can now be **removed from proc.c**:
   - the initcodec function in main
   - the ICR and IER usage in main
   - the MCBSP call in main()

10. **Replace** the setup of the QUE messages and the initial two QUE\_put() operations with 4 SIO\_issue() to ‘prime’ the streams. Note: the size arguments are in bytes!

11. **Replace** the SEM\_pend and QUE\_get API with with an SIO\_reclaim() for each stream. note: you’ll need to 'cast' the return pointers as (Ptr *)pIn and (Ptr *)pOut to match the SIO generic pointer type to your use of short* on pIn and pOut
12. **Replace** the final **QUE_put** at the end of the while loop with a **pair of SIO_issue()s**

*Open myWork.tcf and make the following changes:*

13. The task in the echo lab invoked the audio pass-through function **Audio_echo_Task**
   *Modify the TSK object to invoke procBuf*

14. Add an **IDL object** to invoke the **readDipSwitches** function

15. As the new project has no need for a periodic function, **remove** the PRD function **PRD10000**

16. In order for the in and out buffers to be located in on-chip ram, right click on **MEM - Memory Section Manager** in the **System** folder, select **Properties**, and under the **Compiler Sections** tab, route **.bss** and **.far** to **IRAM** via the drop boxes

17. **Save** the file and close the config tool

18. **Build** the project and download the code. **Verify** the functionality of the system and debug as necessary.

**D. Optional: “Clean” Initial Output Buffer Version**

Time and interest permitting, adapt solution per the coding example of slide 26 in the presentation notes (Double Buffer Stream TSK Coding Example), which eliminates the output of initial spurious buffers. As above, begin with one of the prior solutions above, and if desired, save the results to **C:\BIOS\mySols\06D**.

**E. Optional: Multi-Buffer Versions**

Time and interest permitting, adapt solution per the coding example of slides 27-28 in the presentation notes (Triple/N Buffer Stream Coding Example), which allow for 3 or more buffer solutions. As above, begin with one of the prior solutions above, and if desired, save the results to **C:\BIOS\mySols\06D**.
Multi-Threaded Systems

Introduction

In this chapter, the presence of multiple threads within a system will be considered. Ways to control how the scheduler operates will be considered. In addition, the ability within DSP/BIOS to pace threads by time, rather than data availability, will be considered.

Objectives

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

• Describe the way BIOS can implement a time base
• Setup a time base via the BIOS CLK module
• Describe the results of invoking various BIOS CLK API
• Set functions to run at a periodic rate via the PRD module
• Describe how to implement delayed ‘one-shot’ functions
• Describe how the scheduler can be managed via BIOS API
• List various BIOS scheduler management API
• Select and incorporate scheduler management API to obtain desired performance in a given system

Module Topics

Multi-Threaded Systems .......................................................................................................................... 7-1

Clock Manager: CLK .............................................................................................................................. 7-2

Periodic Functions: PRD .......................................................................................................................... 7-3

Scheduler Management API .................................................................................................................... 7-6
  HWI Management API ......................................................................................................................... 7-6
  SWI Management API ......................................................................................................................... 7-7
  TSK Management API ......................................................................................................................... 7-9

Lab 7: Multi Threaded Systems .............................................................................................................. 7-12
  A. Add a New Thread to the Audio System ......................................................................................... 7-14
  B. Alternate Version of the Multi-Threaded System ............................................................................ 7-15
Clock Manager: CLK

**BIOS Clock Services – CLK API**

<table>
<thead>
<tr>
<th>CLK API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK_gettime</td>
<td>Get low-resolution time (32-bit value)</td>
</tr>
<tr>
<td>CLK_gethtime</td>
<td>Get high-resolution time (32-bit value)</td>
</tr>
<tr>
<td>CLK_getprd</td>
<td>Get period register value</td>
</tr>
<tr>
<td>CLK_countspms</td>
<td>Get number of hardware timer counts per millisecond</td>
</tr>
</tbody>
</table>

- CLK abstracts details of HW timer to provide low-res time / system tick
- Timer period is set and CLK objects specified in BIOS configuration
- CLK can drive periodic objects directly, or at different rates as PRD SWI
- CLK time values are often helpful in real-time analysis (next module)

**Setup of CLK via Configuration Tool**

Setup of the CLK Module
1. right click on CLK mgr
2. select “Properties”
3. define Low res clock rate via usecs/int
4. optionally, set other parameters as desired

Optional:
Making a new CLK object
1. right click on CLK mgr
2. select “Insert CLK”
3. type CLK name
4. right click on new CLK
5. select “Properties”
6. type function to run

All CLK objects are invoked each Lo Res tick – PRD fxns can run at different intervals – next...
Periodic Functions: PRD

DSP/BIOS Periodic Functions

- A special SWI that provides non-preemptive scheduling for periodic functions
- While SIO indicates data available and SEM indicates posting by other thread, when time is the gating event PRD is most ideal choice
- Also useful for modeling interrupts to simulate peripherals (IO devices)

Periodic Events – PRD SWI

- PRD_tick() is invoked by PRD_clock by default (also TSK_tick)
- PRD_tick() may be called by any desired user function as well
- PRD_tick() launches the PRD_swi which
  - Scans the list of PRD_obj’s
  - Determines if the specified time for the given PRD_obj has elapsed
  - If so, the function associated with the PRD_obj is called
- All PRD_obj functions must complete within ONE system (PRD) tick
  - Recommended: make PRD_swi highest priority SWI
  - If routines are short and tick is long - no problem
  - Long functions can be broken up with posts of other threads
Setup of PRD via Configuration Tool

Creating a PRD
1. right click on PRD mgr
2. select “Insert PRD”
3. type PRD name
4. right click on new PRD
5. select “Properties”
6. indicate desired
   • period (ticks)
   • mode
   • function
   • arguments

◆ A PRD can directly launch a regular SWI by specifying:
   • function: _SWI_post
   • arg0: _mySWI
   allowing control of priority, and meeting requirement for
   all PRDs to complete before the next PRD tick

TCONF Setup of PRD Module & Object

PRD.OBJMEMSEG = prog.get("myMEM");
PRD.USECLK = "true";
PRD.MICROSECONDS = 1000.0;

var myPrd = PRD.create("myPrd");
myPrd.period = 1024;
myPrd.mode = "continuous";
myPrd.fxn = prog.extern("_myFxn");
myPrd.arg0 = 0;
myPrd.arg1 = 0;

* Underlying interrupt rate is largest binary number divisible into period
  value, so for lowest overhead, pick a binary number when possible
One-shot Periodic Functions

- Allows delayed execution of a function by $n$ system ticks
- PRD_start() invokes each iteration of the one-shot function
- PRD_stop() can be used to abort a one-shot prior to timeout
- Example of use: software watchdog function

```
PRD Object X
Period 4
Function funcX()
Type 1 shot
Arg0 0
Arg1 0
```

Low-res clock (incremented by system tick)

```
... 74 75 76 77 78 79 80 81 ...
```

```
PRD_start()        funcX()
```

PRD API Review

<table>
<thead>
<tr>
<th>PRD API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRD_tick</td>
<td>Advance tick counter, dispatch periodic functions</td>
</tr>
<tr>
<td>PRD_start</td>
<td>Arm a periodic function for onetime execution</td>
</tr>
<tr>
<td>PRD_stop</td>
<td>Stop a periodic function from execution</td>
</tr>
<tr>
<td>PRD_getticks</td>
<td>Get the current tick counter</td>
</tr>
</tbody>
</table>

- Tick counter can be manually incremented by the user with PRD_tick()
- One-shot periodic functions are managed with PRD_start() & PRD_stop()
- Inspection of tick count is possible with PRD_getticks()

- Continuous periodic functions are set up via the BIOS configuration tool and are generally not managed at run-time via BIOS API
Scheduler Management API

- Generally, threads are automatically managed by BIOS according to the priorities of each thread.
- Sometimes, however, it is desirable to alter the normal BIOS scheduler operation, for example:
  - When deadlines are approaching a thread can temporarily be given higher priority or even exclusive use of the processor.
  - When multiple threads share a resource, priorities can be modified to avoid higher priority threads interrupting critical sections of lower priority threads.
  - To implement time slicing amongst equal priority threads (equal threads are normally “FIFO” serviced).
  - To allow TSKS to ‘sleep’ for a time.
- In these cases, API can be invoked to alter the behaviour of the scheduler with respect to HWI, SWI, and TSK as required.

HWI Management API

**HWI_disable and _restore API**

```c
oldCSR = HWI_disable();
// “critical section” ...
// scheduler inhibited ...
HWI_restore(oldCSR);
```

- **HWI_disable()** Creates a period where no asynchronous events may occur.
- Interrupts that come in during this period will be held off until HWI is re-enabled (if a given interrupt occurs more than once in this period, the additional events will be lost).
- **HWI_restore()** does not necessarily enable interrupts, but instead asserts to state prior to HWI_disable().
**HWI and IDL Scheduler API**

<table>
<thead>
<tr>
<th>HWI, IDL API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWI_enable</td>
<td>Globally enable hardware interrupts</td>
</tr>
<tr>
<td>HWI_disable</td>
<td>Globally disable hardware interrupts</td>
</tr>
<tr>
<td>HWI_restore</td>
<td>Restore global interrupt enable state</td>
</tr>
<tr>
<td>IDL_run</td>
<td>Make one pass through idle functions*</td>
</tr>
</tbody>
</table>

* Not commonly used, not callable by HWI or SWI

**SWI Management API**

**Disabling & Enabling Software Interrupts**

```c
SWI_disable();

// “critical section”...
// SWI scheduler inhibited ...

SWI_enable();
```

- Similar to HWI_disable/_restore
- Concludes with SWI_enable (not “SWI_restore”)
- Acts on SWI scheduling only – HWI continue unchanged
- Nestable - number of levels managed by BIOS
Temporary Elevation of SWI Priority

- `SWI_raisepri()` cannot lower priority (actually disables lower priority levels)
- Priority returns to the original value when the SWI exits
- Original Priority ("origPrio") should be a local variable
- Priority values are bit positions, not integer numbers (eg: priority 7 would be ...0100 0000 b)
- To elevate a SWI above one (or several other) SWI, use in conjunction with `SWI_getpri`, as per the example below:

```c
origPrio = SWI_raisepri(1<<7);
    // critical section ...
    // lower prio SWIs inhibited ...
    SWI_restorepri(origPrio);
```

For Priority level "X" select `1<<X` as the argument to `raisepri`

SWI Scheduler API

<table>
<thead>
<tr>
<th>SWI API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_disable</td>
<td>Disable software interrupts</td>
</tr>
<tr>
<td>SWI_enable</td>
<td>Enable software interrupts</td>
</tr>
<tr>
<td>SWI_getpri</td>
<td>Return an SWI's priority mask</td>
</tr>
<tr>
<td>SWI_raisepri</td>
<td>Temporarily raise an SWI's priority</td>
</tr>
<tr>
<td>SWI_restorepri</td>
<td>Restore an SWI's priority to object value</td>
</tr>
<tr>
<td>SWI_self</td>
<td>Return address of SWI's object</td>
</tr>
</tbody>
</table>
TSK Management API

Disabling & Enabling Task Scheduling

- Similar to SWI_disable/enable
- Acts on TSK scheduling only – SWI & HWI continue unchanged
- Nestable - number of levels managed by BIOS

```c
TSK_disable();

// "critical section" ...
// TSK scheduler inhibited ...

TSK_enable();
```

Modification of a Task’s Priority

```c
origPrio = TSK_setpri(TSK_self(), 7);

// critical section ...
// TSK priority increased or reduced ...

TSK_setpri(TSK_self(), origPrio);
```

- `TSK_setpri()` can raise or lower priority
- Return argument of `TSK_setpri()` is previous priority
- New priority remains until set again or TSK is deleted and re-created
- TSK priority is an integer value: 1 to 15 (unlike SWI, using binary weighted numbers)
- To suspend a TSK, set its priority to negative one (-1)
  - Suspended TSK not part of BIOS TSK scheduling queue
  - TSK can be activated at any time (by some other thread) via `TSK_setpri()`
  - Handy option for statically created TSKs that don’t need to run right away
  - A TSK can be suspended at any time under BIOS, by itself or another thread
TSK_yield : Time Slicing

- TSK_yield() instructs the BIOS scheduler to move the current TSK to the end of the priority queue
- If another TSK of equal priority is ready, it will then be the active TSK
- This API can be invoked at any time by the active TSK or any SWI/HWI
- If a PRD calls TSK_yield, time slicing amongst equal priority TSKs is achieved

TSK_sleep and TSK_tick

- TSK_sleep(Uns sleeptime)
  - Blocks execution of current TSK for n TSK ticks
- TSK_tick()
  - Similar to PRD_tick for PRD SWIs
  - Advances the task alarm tick by one count
  - Default - called from PRD_clock (system tick)
  - If 'ticks' are events and not time, TSK_tick can be called from any thread
  - TSK_itick() is for use inside ISRs w/o dispatcher
Scheduler Management API

**Task Control Block Model**

- BIOS Startup
- READY
  - TSK_yield()
  - TSK_setpri()
- RUNNING
  - TSK_sleep()
  - TSK_yield()
  - SEM_post()
- BLOCKED
  - TSK_tick()
  - SEM_pend()
- TERMINATED
  - TSK_exit()
  - TSK_sleep()
  - SEM_pend()

**TSK Scheduler API**

<table>
<thead>
<tr>
<th>TSK API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSK_disable</td>
<td>Disable DSP/BIOS task scheduler</td>
</tr>
<tr>
<td>TSK_enable</td>
<td>Enable DSP/BIOS task scheduler</td>
</tr>
<tr>
<td>TSK_self</td>
<td>Returns address of task object</td>
</tr>
<tr>
<td>TSK_getpri</td>
<td>Get task priority</td>
</tr>
<tr>
<td>TSK_setpri</td>
<td>Set a tasks execution priority</td>
</tr>
<tr>
<td>TSK_yield</td>
<td>Yield processor to equal priority task</td>
</tr>
<tr>
<td>TSK_sleep</td>
<td>Delay execution of the current task</td>
</tr>
<tr>
<td>TSK_tick</td>
<td>Advance system alarm clock</td>
</tr>
<tr>
<td>TSK_itick</td>
<td>Advance system alarm clock (ISR)</td>
</tr>
<tr>
<td>TSK_time</td>
<td>Return current value of system clock</td>
</tr>
</tbody>
</table>
Lab 7: Multi Threaded Systems

In this lab the solution from the previous lab will be the starting point. To it, a new thread of activity will be added – a periodic event running a simple load function. This load, which performs no valuable function, is used to represent the effect of any useful DSP activity on another real-time thread – in this case, the existing audio thread (the FIR filter).

The lab procedure will include the following activities:

- Open CCS and the solved project from Lab 6
- Add Load.c and NopLoop.asm to the project
- In the TCF file, set the CLK to a .1 second rate and call the load function via a periodic SWI
- Build, run, and test the project; note the quality of results with various DIP switch settings
- Modify the solution to prevent the load from preempting the audio thread
  (see part B on page 15 for hints and/or procedure on how to do this)
- Rebuild, run, test; compare results to prior lab

This lab can be attempted using only the information on these first two pages. In most cases, it is recommended to follow the steps on the following pages. Given the experience gained in prior labs, the procedures on the next page are briefer and more demanding. Refer to prior labs or ask the instructor for assistance if you do not know how to implement steps in the procedure that follows.

On the next page are two images. The first depicts the overall layout of the lab. On the left side of the diagram is the audio system seen previously. On the right is a new thread – the load function, triggered by the BIOS managed clock as a periodic SWI. The load function provides a bit of user interface: DIP switches 2 and 3 control the amount of load presented to the CPU, and the LEDs are used to indicate each call of the load function and the current level of the load. Below this is a listing of the key code components of the new files that will be added to the prior lab to create this multi-threaded example. In Load.c the DIP switches are monitored to determine the desired load amount, the corresponding LED is flashed, and the load function is called with a load amount based on the current DIP settings. Another file, load_6416.asm, is also required, which implements the actual dummy load function. This function was written in assembly so that the optimizer would not eliminate its effect when the release version of the system is tested.

When this system is created and tested, any problems found should be resolved, using the techniques made available with DSP/BIOS.

In this lab, the focus should be on the use of DSP/BIOS in managing the complications introduced when additional threads are added to a single threaded solution. While a quick review of the load files is an option, it should not represent any significant investment of time here, since the load is not the point of the exercise, and doesn’t demonstrate any really beneficial DSP work.

As noted, the new files needed for this lab will be found in C:\BIOS\Algos. If needed, solution files are in C:\BIOS\Sols\07. If this particular chapter is of lesser interest, or if there is a time constraint, you may instead copy the solution files to the \Work directory, skip over the authoring steps, and move directly to seeing how the completed code looks and works.
Lab 7a: Multiple Threads

- Begin with Lab 6 solution
- Add Load.c and NopLoop.asm (Algos dir) to project
- In TCF file: set up CLK rate, create PRD SWI running at 100mSec rate, calling fxnLoad
- Build, load, run; test audio w. range of DIP cases

Load.c

```c
void fxnLoad(void) {
    short i;
    unsigned char mask, dips;
    static Bool blink = 0;
    EVMDM6437_I2C_read( I2C_GPIO_GROUP_0, &dips, 1 );
    if(hw_sw0 == (dips>>4&1)) {sw0 = hw_sw0 = !(dips>>4&1);} 
    if(hw_sw1 == (dips>>5&1)) {sw1 = hw_sw1 = !(dips>>5&1);} 
    if( ( hw_sw2==(dips>>6&1)) | (hw_sw3== (dips>>7&1)) ){
        sw2 = hw_sw2 = !(dips>>6&1);
        sw3 = hw_sw3 = !(dips>>7&1);
    }
    switch(2*sw2+sw3){
        case (3) : for (i=0; i< 7; i++) {load(5000);}
        case (2) : for (i=0; i<12; i++) {load(5000);}
        case (1) : load(5500);
        case (0) : load( 100);
    }
    blink^=1;
    mask = (char)( 0x0F^((1^blink)<<2*sw2+sw3) );
    EVMDM6437_I2C_write( I2C_GPIO_GROUP_1, &mask, 1 );
    IDL_run();
}
```
**A. Add a New Thread to the Audio System**

1. Start **CCS. Load** (and verify the performance of) the solution from Lab 6C

2. Copy **NopLoop.asm** and **Load.c** from the **Algos** to the **Work** directory, then **add them to the project**. The C file inspects DIP switches 2 and 3 and blinks the LED corresponding to the settings of these switches. In addition, it calls a looping no-op function in the .asm file that simulates the effect of a competing thread added to the audio application.

   *In the config file:*

3. Set the **CLK - Clock Manager** properties to a **100,000 microSecond** (0.1 second) rate.

4. Open the SWI manager folder, note that the **KNL_sw1** is the sole entry.

5. In the PRD function manager **add a PRD object**, the default name **PRD0** will be OK (you can rename if you like). Note that a new entry "PRD_sw1" appears in the SWI listing to reflect usage of periodic SWIs.

6. **Remove** the **readDipSwitches** activity from IDL and **dipMonitor.c** from the project.

7. Set the period of **PRD0** to **1 tick** (0.1 second rate) and function as **_fxnLoad**

8. **Scan the _fxnLoad function in Load.c.** It consists of two parts. The first is a test to see if DIP switches 2 or 3 have been changed (much like the idle thread in DipMonitor.c that we've been using to manage switches 0 and 1 in all prior labs). The 2nd part calls the assembly-authored 'dummy load' function that simulates a CPU load whose duration is determined by the settings of switches 2 and 3, and also blinks the LED corresponding to the switch setting each time the routine is run. While it performs no useful DSP activity, it is representative of the effect some other thread would have on our audio thread.

9. **Build/load, and/run** the project in debug version. The music should play normally with the DIP switches furthest from the USB and power jacks in the up position. The load function's operation should be indicated by the furthest LED blinking.

10. **Test all settings of switches 3 and 2** – note which LED is blinking, the addition of more load in the CPU Load graph, , and any interreference in the audio quality (most noticeable with the LPF running).

11. **Rebuild in release mode** and see if the audio quality with switch 3 on improves. Note your observations and hypothesis for what was observed.

__________________________________________________________________________

12. Based on the BIOS concepts presented to this point, how do you think the affect on the audio thread can be eliminated?

__________________________________________________________________________

13. **Optional:** save the files in C:\BIOS\Labs\Work to C:\BIOS\mySols\07
B. Alternate Version of the Multi-Threaded System

The periodic function added in this lab was invoked as a SWI. Since all SWIs have priority over all TSKs, the audio thread was preempted by the load, and the audio quality is compromised when more than a minor load is present. A number of options exist for resolving this problem. Larger buffers could be applied to the FIR filter to provide longer deadlines. Similarly, more buffers could be added to the streams to add more units of deadline response time. Both of these are a valid and important option when tuning complex multi-threaded systems, since they allow lower priority threads to have a chance to meet their deadlines even though they are vulnerable to pre-emption by higher priority threads. However, in this system, a simpler solution can be sought: changing the relative priority of load and audio. But, if all SWIs outrank all TSKs, how can this be done? PRDs invoke SWIs, so this is fixed, and TSKs are better suited to SIO, so these would be best kept this way. Is there another option? What if the PRD posted a SEM, and the load function (fxnLoad) was within the while loop of a TSK pending on the PRD’s SEM - would this be a possible solution to changing the relative priorities of the two threads? If so, what additional detail must we assure to complete this solution? Note your answer here:

__________________________________________________________________________

Make the changes noted above in the project and determine if the temporal deadlines are now being met. Experiment with all the different load factors in debug and release versions. Refer to prior chapters for assistance with the details of making these changes, if required. Note your observations:

__________________________________________________________________________

__________________________________________________________________________

If needed, refer to the steps on the next page on how to implement this part of the lab.
1. Create a new SEM named `semLoad`
2. Change the `PRD` to invoke `_SEM_post` (instead of `fxnLoad`), with an arg0 of `semLoad`
3. Pend on `semLoad` within the while loop of `tskLoad` in the file `Load.c`
4. Call `fxnLoad` after the pend
5. Create a new TSK, calling `_tskLoad`
6. Build, load, run, and test the system in both debug and release modes. Did making load a TSK allow the audio thread to run without interference? What else might need to be done? Consider their priorities
7. Click on the TSK manager icon. In the right pane, raise `tskProcessBuffer` to priority 2 and leave the new task at priority 1, making the audio thread higher priority than the load.
8. Build, load, run, and test the system. Debug as necessary.
9. Save the contents of C:\BIOS\Labs\Work to C:\BIOS\mySols\07b
Introduction

In this chapter a number of BIOS features that assist in debugging real-time system temporal problems will be investigated.

Objectives

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

- Demonstrate how to obtain statistical data on variables without halting the DSP
- Describe why printf() is unsuitable for real-time systems
- Describe how LOG_printf() overcomes this problem
- Demonstrate how to use LOG_printf() in debugging
- Describe how to implement trace control
- Demonstrate how to perform real-time graphing
- Describe the various API for responding to system errors

Module Topics

- BIOS Instrumentation .............................................................................................................................. 8-1
- Concepts.................................................................................................................................................. 8-2
- Statistics – STS....................................................................................................................................... 8-3
- Data Logging: LOG................................................................................................................................... 8-8
- Trace Control: TRC................................................................................................................................. 8-13
- Instrumentation Overhead ....................................................................................................................... 8-14
- Real-Time Graphing .................................................................................................................................. 8-15
- System Errors: SYS.................................................................................................................................... 8-16
- Lab 8: Instrumentation ......................................................................................................................... 8-19
  A. STS – Activity Monitor ....................................................................................................................... 8-19
  B. STS - Data Monitor............................................................................................................................... 8-20
  C. STS - Temporal Monitor....................................................................................................................... 8-20
  D. Implicit Task Statistics......................................................................................................................... 8-21
  E. LOG – Event Reporting......................................................................................................................... 8-21
  F. (Optional) Trace Control "TRC"............................................................................................................. 8-22
  G. (Optional) SYS - System Error Management...................................................................................... 8-22
Stop-Based Debug Tools

- **Watch Window**
  - Non-real-time technique for checking if results are correct
  - Host needs to interrupt target or program needs to hit a breakpoint to update values

- **Profile points**
  - Inturrupts target to read and pass values to host
  - Interferes with real-time performance of a system

- **How can data be observed without interfering with real-time performance?**
Statistics – STS

Data Visibility w/o Loss of Real-Time

- Data is accumulated on the target in hard real-time
- Data is sent to host in soft real-time
- Target is not halted during critical events to send data

Note: Data is always accumulated & occasionally uploaded

Counting Event Occurences: STS_add

```c
#include <std.h>
#include <sts.h>
myFunction()
{
    STS_add(&myStsObj,NULL);
    ...
}
```

- Putting `STS_add()` in any thread allows the number of times the thread ran to be counted by BIOS
- `myStsObj` can be created in the config tool
- Any number of statistical objects can be created
- *This ability is automatically provided for all SWIs!*
For the Average Value of a Variable

```c
#include <std.h>
#include <sts.h>
myFunction()
{
    STS_add(&myStsObj, myVar);
    ...
}
```

- Note the addition of `myVar` as the 2nd argument to the `STS_add` API.
- This directs BIOS to read this variable each time `STS_add` is run and add it into the running count maintained in the STS object's "Total" register.
- The host PC calculates and displays the "Average" as the "Total" divided by the "Count" (note: this is not done on the DSP).
- In addition, the host display can display the average scaled by user selected sums, products, and divisors.

Statistical Maximum Register Usage...

- Track the maximum and average for a variable is the same API as obtaining the average value. The STS object maintains the max value separately in the 'maximum' register.

```c
STS_add(&myStsObj, value);
```

- Tracking minimum value of a variable is the same as the maximum, except that "- value" (minus value) is specified:

```c
STS_add(&myStsObj, -value);
```

- To monitor the difference between actual values and a desired value, the following can be used:

```c
STS_add(&myStsObj, abs(value-DESIRED));
```
**STS_set() and STS_delta()**

Timing events or monitoring incremental differences in a value
- Uses unsigned arithmetic for subtraction to prevent the problem when value “wraps”
- STS_set() could be placed early in an HWI and STS_delta() could be at the end of an SWI to measure total response time to an interrupt (min/max/average)
- STS_set() and STS_delta() can also be used to monitor growth in a data value between two points in code

```c
STS_set(&myStsObj, CLK_gethtime() );
// algorithm or event to measure...

STS_delta(&myStsObj, CLK_gethtime() );
```

BIOS provides an implicit STS for TSKs that are ‘set’ when a pending TSK is made ready (unblocked). Other API that act on the implicit TSK STS object:
- TSK_deltatime() – to determine the time since the ‘set’ (unblock) of the TSK
- TSK_settime() – to initialize the ‘Previous’ value prior to the TSK while loop
- TSK_getsts() – copies implicit TSK STS obj for any desired user inspection

**STS API**

<table>
<thead>
<tr>
<th>API</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS_add</td>
<td>Add a value to a statistics accumulator</td>
</tr>
<tr>
<td>STS_delta</td>
<td>Add computed value of an interval to accumulator</td>
</tr>
<tr>
<td>STS_reset</td>
<td>Reset the values in the STS object</td>
</tr>
<tr>
<td>STS_set</td>
<td>Store initial value of an interval to accumulator</td>
</tr>
</tbody>
</table>
Statistics Accumulators Objects

- Initial value (32 bit)
- Format used to display the data from object
- Filter Operation
  - Values for A, B, C

<table>
<thead>
<tr>
<th>Target</th>
<th>Host</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Count</td>
<td>Count</td>
</tr>
<tr>
<td>Total</td>
<td>Total</td>
<td>Total/Count</td>
</tr>
<tr>
<td>Maximum</td>
<td>Maximum</td>
<td>Average</td>
</tr>
<tr>
<td>Previous</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Setting Up an STS Object

Creating an STS object
1. right click on STS mgr
2. select “Insert STS”
3. type STS name
4. right click on new STS
5. select “Properties”
6. indicate desired
  - Previous value
  - Unit type
    - Not time based
    - High/Low resolution time based
  - Host operation – equation to scale statistical results by in a variety of equation values and formats as shown...
HWI Monitor Option – STS Collection

- STS optionally available with HWI, as per the properties dialog box below
- Useful to monitor SP, TOS, register, data value (specify address / label)
- Not default to avoid cycle overhead on HWI

Viewing the Count Value via CCS

- Open the STS display via: DSP/BIOS | “Statistics View”
- Right click on the statistics window and select “properties page” to tune the display
Data Logging: LOG

`printf()`

- **Host**: CCS
- **Target**: DSP

```plaintext
ISR {
  ...
  x=function();
  printf("x=%d",x);
  ...
}
```

- ***printf() is a long function* – over 10K
- ***Slow to complete* – uses MIPS, often at critical times
- ***Alters R/T execution rate!***
- Tends to be removed for production
  - Hard to field test
  - Differs from version tested

---

**LOG_printf()**

- **Host**: CCS
- **Target**: DSP

```plaintext
ISR {
  ...
  x=function();
  LOG_printf(trace, "x=%d",x);
  ...
}
```

- ***LOG_printf() is a small function* – @ 100 words
- ***Fast to complete* – assembly authored
- ***Preserves R/T execution rate!***
- Remains in place for production
  - Field test identical to lab environment
  - Identical to version tested

Removing `printf()` can actually save more space than required for all of BIOS!
### printf() vs. LOG_printf()

<table>
<thead>
<tr>
<th>Contained in:</th>
<th>printf()</th>
<th>LOG_printf()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formats data with:</td>
<td>RTS Libraries</td>
<td>DSP/BIOS</td>
</tr>
<tr>
<td>Passes data to host in:</td>
<td>Hard real-time</td>
<td>Soft real-time</td>
</tr>
<tr>
<td>Written in:</td>
<td>C</td>
<td>ASM</td>
</tr>
<tr>
<td>Deterministic:</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimization:</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### LOG_printf() and LOG_event()

**LOG_printf(hLog, format, arg0, arg1)**
- Writes to system or user log as specified by hLog
- ‘format’ argument is pointer to a fixed string & stored on host
- One or two values can be specified to store in log buffer
- These values can be displayed as decimal, hex, or ptr to fixed string

**LOG_event(hLog, arg0, arg1, arg2)**
- Similar to LOG_printf()
- Additional argument replaces ptr to format string
- Allows a bit more data to be recorded

```c
#include <std.h>  // must be 1st include
#include <log.h>  // allow LOG API
extern far LOG_Obj hMyLog;  // refer to GCONF LOG obj
func(){
    LOG_printf( &hMyLog, "X = %d Y = %d", x, y );
}
```
Log Buffer Details

- If data is exported to host faster than it is being logged, all data is observable.
- If not, user can increase update rate or buffer size.
- If transfer lags collection rate, user has choice of how buffer fills:
  - Fixed: once buffer fills, collection stops
    - Optimal to get 1\textsuperscript{st} N samples after an event
    - Useful: LOG\_reset to clear buffer
  - Circular: buffer continues to fill, old data overwritten
    - Optimal for maintaining 'last N' samples
    - Most recent data is always present to view
- Circular/fixed option available via GCONF, TCONF
- Sequence number makes it possible to know the start/end point of circular buffers
- Sequence number is managed intrinsically by BIOS

Other LOG API

**LOG\_message(format, arg0)**
- Writes to system log
- Conditional only to global TRC bits: TRC\_GBLHOST, TRC\_GBLTARG
- Identifies key info during debug

**LOG\_error(format, arg0)**
- Writes to system log
- Not conditional to TRC bit status
- Generates assertion mark on execution graph
- Useful for recording critical system errors

**LOG\_disable(hLog)**
- Halts logging of data for specified log

**LOG\_enable(hLog)**
- Restarts logging of data for specified log

**LOG\_reset(hLog)**
- Discards any prior data in specified log
### LOG API Review

<table>
<thead>
<tr>
<th>LOG API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_print</td>
<td>Add up to 2 values + string ptr to a log</td>
</tr>
<tr>
<td>LOG_event</td>
<td>Add 3 values to a log</td>
</tr>
<tr>
<td>LOG_error</td>
<td>Write a value and string ptr to sys log unconditionally</td>
</tr>
<tr>
<td>LOG_message</td>
<td>Write a value and string ptr to sys log if global TRC bits enabled</td>
</tr>
<tr>
<td>LOG_reset</td>
<td>Discard values in log</td>
</tr>
<tr>
<td>LOG_enable</td>
<td>Start collecting values into log</td>
</tr>
<tr>
<td>LOG_disable</td>
<td>Halt collecting values into log</td>
</tr>
</tbody>
</table>

The LOG module captures information in real-time.

### LOG Setup Via GCONF and TCONF

1. right click on LOG mgr
2. select “Insert LOG”
3. type LOG name
4. right click on new LOG
5. select “Properties”
6. indicate desired
   - Memory segment for buffer
   - Buffer length
   - Circular or fixed
   - Datatype – printf or raw data
   - If raw, select display format

```c
LOG.OBJMEMSEG = prog.get("IRAM"); // place logs objects in "IRAM"
var myLog = LOG.create("logDipSw"); // create new log obj 'logDipSw'
myLog.comment = "DIP sw Monitor"; // comment field – anything you like...
myLog.bufSeg = prog.get("IRAM"); // specify memory to draw buffer from
myLog.bufLen = 16; // size of log buffer array – binary number
myLog.logType = "circular"; // or "fixed"
myLog.dataType = "printf"; // for LOG_print, "raw data" for LOG_event
myLog.format = "0x%x, 0x%x, 0x%x"; // log data display formats, also: %o, %s, %r
```
Setup Message Logging

To view logs:
- DSP/BIOS -> Event Log
- Select logs to view

To write logs to files
- Right click on message log
- Open the Property page
- Select “Log to File”
- Specify file name
- Collect data
- Close message log window to view file
Trace Control: TRC

Host (CCS) Trace Control

- Control Panel can be used to manage target trace control bits
- Implicit Instrumentation options
- Explicit (user) Instrumentation – for any general purpose control
- Changeable on target only...
- Global Enable – allow control of overall instrumentation without changing individual settings

Target (API) Trace Control

```
#include <trc.h>  // include TRC API
...
TRC_disable(TRC_LOGCLK);  // Inhibit clock logging
...
TRC_enable(TRC_LOGCLK | TRC_LOGPRD);  // Turn on clk & prd logging
...
if (TRC_query(TRC_USER0) == 0) {  // log if user0 is set
    LOG_printf(&logObj, ...);
}
...
if (TRC_query(TRC_USER1 | TRC_STSPRD) == 0) {  // multi condition ex.
    STS_delta(&stsObj, CLK_gethtime());
}
```

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRC_LOGCLK</td>
<td>Disable specified trace control(s)</td>
</tr>
<tr>
<td>TRC_LOGPRD</td>
<td>Enable specified trace control(s)</td>
</tr>
<tr>
<td>TRC_LOGSWI</td>
<td>Returns enable/disable status of specified trace control(s)</td>
</tr>
</tbody>
</table>
RTA Properties: Set Refresh Rate

To set update rate of RTA events
- DSP/BIOS -> RTA Control Panel
- Right click on the RTA Control Panel
- Select the Property Page
- Allows setting of rates for STS, LOG, RTA

Instrumentation Overhead

- Cycle overhead for various instrumentation API

<table>
<thead>
<tr>
<th>API</th>
<th>54xx</th>
<th>55xx</th>
<th>6xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_printf, event</td>
<td>30</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>STS_add</td>
<td>30</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>STS_delta</td>
<td>40</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>TRC_enable, disable</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

- Code size increase with instrumentation
  - 5000 systems: ~2K (nmadu's)
  - 6000 systems: ~7K (nmadu's)

- Kernel can be built without instrumentation by unchecking the “Enable Real Time Analysis” option in the Global Settings module of the Configuration Tool

- Low code size increase allows image released to be exactly the one tested (“Test what you fly, fly what you test...”)
Real-Time Graphing

BIOS Visual Real-Time Analysis

- CPU Load Graph - Shows Overall Processor Load
- Execution Graph - Shows Preemption amongst threads
  event based – add PRD_tick to add a time scale

Kernel Aware Debugger

Useful for debugging your application
- Provides insight into DSP/BIOS Objects
- Useful for determining what state a TSK, SWI, SEM, etc are in
- Updates at breakpoint or at user request
System Errors: SYS

SYS_error() and SYS_abort()

SYS_error(string,errno,[optarg], …);
- Used to indicate errors in application programs or internal functions
- Called by DSP/BIOS and by user-written modules
- Default function _UTL_doError logs an error message

```c
buf = (Ptr)MEM_calloc(0, BUFSIZE, BUFALIGN);
if (buf == NULL) {
    SYS_abort("Error: MEM_calloc failed.");
    // exit & report failure
}
```

SYS_abort(format, [arg,] …);
- Used for unrecoverable errors
- Calls the function bound to the abort function
- Default function _UTL_doAbort
- User can specify a function to use

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_UTL_doError</td>
<td>logs an error message and returns</td>
</tr>
<tr>
<td>_UTL_doAbort</td>
<td>logs an error message and calls _UTL_halt</td>
</tr>
<tr>
<td>_UTL_halt</td>
<td>disables interrupts and enters infinite loop</td>
</tr>
</tbody>
</table>

```c
void UTL_doError(String s, Int errno)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = '%s'", s);
}
```

SYS Definitions

The following definitions are used in BIOS systems to identify common timeout and error conditions

<table>
<thead>
<tr>
<th>BIOS Symbol definition</th>
<th>#</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define SYS_OK</td>
<td>0</td>
<td>// no error</td>
</tr>
<tr>
<td>#define SYS_EALLOC</td>
<td>1</td>
<td>// memory allocation error</td>
</tr>
<tr>
<td>#define SYS_EFREE</td>
<td>2</td>
<td>// memory free error</td>
</tr>
<tr>
<td>#define SYS_ENODEV</td>
<td>3</td>
<td>// device driver not found</td>
</tr>
<tr>
<td>#define SYS_EBUSY</td>
<td>4</td>
<td>// device driver busy</td>
</tr>
<tr>
<td>#define SYS EINVAL</td>
<td>5</td>
<td>// invalid device parameter</td>
</tr>
<tr>
<td>#define SYS EBADIO</td>
<td>6</td>
<td>// I/O failure</td>
</tr>
<tr>
<td>#define SYS EMODE</td>
<td>7</td>
<td>// bad mode for device driver</td>
</tr>
<tr>
<td>#define SYS EDOMAIN</td>
<td>8</td>
<td>// domain error</td>
</tr>
<tr>
<td>#define SYS ETIMEOUT</td>
<td>9</td>
<td>// call timed out</td>
</tr>
<tr>
<td>#define SYS EEOF</td>
<td>10</td>
<td>// end-of-file</td>
</tr>
<tr>
<td>#define SYS EDEAD</td>
<td>11</td>
<td>// previously deleted obj</td>
</tr>
<tr>
<td>#define SYS EBADOBJ</td>
<td>12</td>
<td>// invalid object</td>
</tr>
<tr>
<td>#define SYS EUSER</td>
<td>256</td>
<td>// user errors start here</td>
</tr>
<tr>
<td>#define SYS FOREVER</td>
<td>-1</td>
<td>// block until posted – no timeout</td>
</tr>
<tr>
<td>#define SYS POLL</td>
<td>0</td>
<td>// zero block time</td>
</tr>
</tbody>
</table>
## BIOS Termination

<table>
<thead>
<tr>
<th>SYS API</th>
<th>System settings management</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_exit</td>
<td>Terminate program execution</td>
</tr>
<tr>
<td>SYS_atexit</td>
<td>Stack an exit handler</td>
</tr>
</tbody>
</table>

**SYS_atexit(handler)**
- Pushes an exit handler function on an internal stack
- Up to eight handlers allowed
- Returns success or failure (full stack)

**SYS_exit(status)**
- Pops any exit handlers registered by SYS_atexit() from internal stack
- Calls all handles and passes status to handler
- Calls function tied to Exit - default is UTL_halt

## SYS printf API

<table>
<thead>
<tr>
<th>SYS API</th>
<th>System settings management</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_putchar</td>
<td>Output a single character</td>
</tr>
<tr>
<td>SYS_printf</td>
<td>Formatted output</td>
</tr>
<tr>
<td>SYS_sprintf</td>
<td>Formatted output to string buffer</td>
</tr>
<tr>
<td>SYS_vprintf</td>
<td>Formatted output variable argument list</td>
</tr>
<tr>
<td>SYS_vsprintf</td>
<td>Output formatted data</td>
</tr>
<tr>
<td>__UTL_doPutc</td>
<td>writes a character to the system trace buffer</td>
</tr>
</tbody>
</table>

- SYS printf functions allow a buffer to be filled in the manner of the C function printf
- The location and size of the buffer are under user control via GCONF or TCONF
- The use of these functions is discouraged, since they are large and slow – like printf
- SYS_putchar, _printf, _vprintf, _sprintf, _vsprintf all fill trace buffer via putc function, both defined via GCONF or TCONF
- The system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the Code Composer Studio memory view)
Setting SYS Properties

- SYS_abort, _error, _exit each call a user defined function, as defined with GCONF or TCONF, as seen below

- SYS_putchar, _printf, _vprintf, _sprintf, _vsprintf all fill trace buffer via putc function, both defined via GCONF or TCONF (The system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the Code Composer Studio memory view)

- UTL functions shown are system defaults, and can be replaced with any other user authored function as desired

TCONF script segment

```
SYS.TRACESIZE = 512;
SYS.TRACESEG = prog.get("IDRAM");
SYS.ABORTFXN = prog.extern("UTL_doAbort");
SYS.ERRORFXN = prog.extern("UTL_doError");
SYS.EXITFXN = prog.extern("UTL_halt");
SYS.PUTCFXN = prog.extern("UTL_doPutc");
```
Lab 8: Instrumentation

The goal of this lab will be to add a variety of instrumentation to the solution from the prior lab, in order to better observe various details of interest in the system as it runs. As shown in this chapter, these instrumentations are non-intrusive, such that these observations of activity will not perturb the real-time activities of the running system.

The lab procedure will include the following activities:

• Begin with the solution to lab07, which should be in C:\BIOS\Labs\Work
• Apply a number of STS objects to report simple through advanced information
• Use LOG_printf() to receive reports of activities during system operation
• Manage what is reported via TRC control
• Identify problems via the use of status return values
• Respond to status failures and report error causes via SYS functions

If time permits, for a more rigorous test of your skills, you can attempt each lab segment based on the concepts in the introductory paragraph alone. In most cases, it is recommended to follow the steps which follow the introduction. Given the experience gained in prior labs, the procedures will be briefer and more demanding. Refer to prior labs or ask the instructor for assistance if required.

To start, open CCS and load myWork.pjt. This should provide a starting point here matching the end point of the prior lab (the multi-threaded audio-plus-load solution).

A. STS – Activity Monitor

One metric that might be useful in the test and debug of various systems would be to see how often a particular function has run. As seen in the lecture materials, an STS object can be applied to perform this measurement. Add an STS to tskLoad to count the number of times this task has run. Build the code and run the system. Open the STS monitor to observe the changing counter value.

1. Start CCS. Open the workspace with the solution from Lab 7.
2. Create a new STS object in the TCF file named eventCounter
3. Add an STS_add() in the while loop of the tskLoad function, with a first argument of the handle to the STS object just created, and second argument of '0'
4. Build, load, and run the program
5. Open the Statistics View under the DSP/BIOS menu to observe the number of times the TSK has run, as indicated by the count value of the STS object just created. Note also the SWIs in the system also in the STS list. All SWIs include intrinsic STS objects, which allows the same monitoring ability added to the TSK to be present on all SWIs in the system - a handy convenience for the system debug engineer!
6. Save results as per prior labs
B. STS - Data Monitor

In addition to knowing how often a thread ran, it is often helpful to know the statistics of key data values in use by the system. STS objects can easily do this, too. To try this, the goal of this build will be to add a new STS object to track the statistics of the input data values.

1. If necessary, start CCS and open the workspace with the prior lab solution.
2. Add another STS object named \texttt{stsAudioData}
3. In the \texttt{for} loop that updates the history buffer in the \texttt{while} loop of \texttt{procBuf} (in \texttt{proc.c}), insert an \texttt{STS\_add()} monitoring \texttt{pIn[i-HIST]}
4. Build, load, and run the program
5. Right click on the STS window and select Property Page. In the General tab, check the newly added STS object to add this one to those being displayed. If you like, uncheck those not of interest to you at this point. Notice how the count grows much faster than that of the \texttt{eventCounter} monitor, since it is counting data samples in a much faster thread. Also note that the average is close to zero, since audio signals are centered around a zero DC offset. Finally, note the max is within the value of a signed 16 bit number. Given this information, it is seen that the data being monitored has the expected characteristics of an audio data stream.

6. Save results as per prior labs

C. STS - Temporal Monitor

Another valuable use of STS is the ability to monitor the time taken between two points in code. In single-thread systems, this value is usually consistent from case to case, however, in multi-threaded systems, preemption and other factors can cause execution time to vary significantly. The ability to measure the execution time of a given thread in a complex system can provide critical information on the reliability of the real-time success of the overall system. In this build, the ability to monitor average and worst-case temporal performance will be implemented.

1. If necessary, start CCS and open the workspace with the prior lab solution.
2. Create another STS object named \texttt{audioTime} and set its Unit Type property to High Resolution Time Based
3. In \texttt{proc.c}, set the STS after the stream reclaim's, and obtain the elapsed time after the \texttt{SIO\_issue}'s. You may need to include another BIOS library for this to build without warning.
4. Build, load, and run the program. Observe the time results. As before, go to the STS Properties page and enable the new STS object. In addition, set the units for the new object to microseconds

Is the execution timing similar from run to run, or does it vary widely? Try changing the load values with the DIP switches and see what affect this has. Right click on the STS display and select "clear" from the menu to reset the object values as required. How closely do the max and average rates compare? What does this tell you about the priority of the task being measured? Reverse the thread priorities and note how this affects execution time. Also, note how building in 'release' version changes these numbers as well.

5. Save results as per prior labs
D. Implicit Task Statistics

In the prior builds, the rate of execution from return of the SIO reclaim until after the SIO issue was measured. This may not be all the time from when the IOM issued a buffer to the stream until the buffer was given back to the IOM. What if the audio TSK was not the highest priority thread when the buffer was posted? The thread would have been moved from the blocked to ready state, but execution would be held off until higher priority threads were served. This can greatly affect the critical measurement of buffer turn-around time, especially when the TSK in question is not the highest priority in the system. Fortunately, BIOS provides an implicit STS object for each TSK, and the equivalent of an STS_set is performed each time the TSK is unblocked - perfect for this case. To observe these statistics, follow the procedure below:

1. If necessary, start CCS and open the workspace with the prior lab solution.
2. Add **below the STS_delta(): TSK_deltatime(TSK_self() ).** The equivalent operation for the STS_set after the SIO reclaims is performed automatically by BIOS, so nothing else needs to be explicitly done to complete the collection of this data.
3. **Build, load, and run.** Again, **add the new STS object to the list of those shown,** and **set the units to microseconds.** Compare the implicit task values with those of the prior STS object. Build again with different TSK priorities and in debug vs release to see how these numbers compare over a range of cases - especially the case where the audio task priority is lower than that of the load, and the load setting switches in the next to highest setting.
4. Make your final build with the audio TSK at higher priority than the load TSK.
5. **Save** results as per prior labs.

E. LOG – Event Reporting

1. If necessary, start CCS and open the workspace with the prior lab solution.
2. **Create a LOG** object named **logDipSw.** The default log properties are sufficient
   In Load.c
3. **Add LOG_printf( &logDipSw, "Load = %d", 2*sw2+sw3 );** at the end of the if() statement testing sw2 and sw3 in fxnLoad. The equation "2*sw2+sw3" will display values 0 thru 3 as the load applied, where larger numbers indicates larger load amounts
4. **Build, load, and run** the program.
5. Under **DSP/BIOS**, open **Message Log.** Right click on the Message Log window and select **Float In Main Window.** Change some of the DIP switch settings and observe the message log reporting on these changes.
6. **Change the log properties to buflen =8 and logtype = fixed.** Rebuild and rerun to see the difference from the prior build. How much data was collected? Why is this so?
7. Change the logtype back to circular with a smaller buflen; build and retest. Can this small circular buffer keep up with the event rate indefinitely, even when rapidly changing the DIP switches? Consider the log sequence number in determining the answer to this question.
8. **Save** results as per prior labs.
F. *(Optional)* Trace Control "TRC"

1. If necessary, start CCS and open the workspace with the prior lab solution.
2. **Right click on the STS display**, select **Properties Page**, and **enable all objects** for display.
3. Open the **DSP/BIOS | RTA Control Panel**. One at a time, **uncheck** the **SWI, PRD, and TSK** accumulators and the **Global host enable** option. Note how each affects the accumulation of statistics of particular objects in the STS window. While the time expended in maintaining these implicit STS objects is small, it is worth noting that these can be inhibited if desired - either via the control panel, as seen here, or via TRC API, as noted in the lecture notes.
4. Add **if (TRC_query(TRC_USER0) == 0)** above the LOG_printf in **Load.c**
5. To make TRC API available to this file add an **include** of **trc.h**
6. **Rebuild** and **retest**. Note the affect now of the **USER0 checkbox** in the RTA control panel on the collection of LOG data.
7. **Save** results as per prior labs.

G. *(Optional)* SYS - System Error Management

Normally, when there are coding errors, the system continues to run. The error causes faults elsewhere in the system, and when it is realized that something is wrong, a lengthy process of backtracking to discover the original error ensues. It would make more sense during debug to halt a session when the error first occurs and to be informed of what happened to cause the system to stop. All this is possible with use of SYS API. The most dramatic of these - **SYS_abort**, will be used here to monitor a potential problem. Let’s imagine that our system could not tolerate all 4 DIP switches being in the down position at once, and if it happened, the audio would fail. Normally there’d be nothing to explain why the audio failed, other than an exhaustive search for the problem. This potential debug time drain can be avoided by the addition of a test for the problem, and the use of **SYS_abort** to halt the system if an error is found. The argument of **SYS_abort** can be a string telling the programmer what occurred, as implemented by the procedure below:

1. If needed, start CCS and load a prior lab project.
2. To create an abort condition when all 4 DIP switches are in the ‘down’ position, in **load.c** follow the I2C_read call with the following code:
   ```c
   if((dips>>4&0xF)==0xF)
       SYS_abort(" All Switches Were Pressed ! ");
   ```
3. **Build, run, and test**. Music should break when all switches are pressed down.
4. Look at the final entry in the **Execution Graph Details** in the **Message Log window**. Note the information provided there on why the system had exited.
5. **Remove the dip test and SYS_abort** and verify the system runs normally again.
6. **Save** results as per prior labs.
Introduction

In this chapter the details of setting up a static DSP system will be considered.

Objectives

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

- List the advantages and limitations of static systems
- Demonstrate how to define target memory in CCS
- Demonstrate how to route software components into desired hardware memory
- Describe the files created in a CCS project build
- Observe the results of a built project
- Describe how to optimally tune a static system
- Describe the startup sequence of a BIOS based system

Module Topics

Static System Design ................................................................................................................................. 9-1

- Concepts.................................................................................................................................................. 9-2
- Hardware Segments................................................................................................................................. 9-4
- Software Sections ................................................................................................................................. 9-5
  - System Stack .................................................................................................................................... 9-5
  - C Sections ........................................................................................................................................... 9-6
  - BIOS Sections ................................................................................................................................. 9-8
  - User Sections ................................................................................................................................. 9-10
- Files Created......................................................................................................................................... 9-11
- Observe & Tune Results ....................................................................................................................... 9-12
- Startup Sequence................................................................................................................................. 9-13
- Lab 9: Static System Management ..................................................................................................... 9-14
  A. Map File Inspection ...................................................................................................................... 9-14
  B. Remapping the Output Buffers ..................................................................................................... 9-14
  C. Remapping the Input Buffers ........................................................................................................ 9-14
  D. Alignment of Buffers for Cache ..................................................................................................... 9-14
## Static System Concepts

- **What is a static system?**
  - One in which all components remain in place during the life of the system
  - No components are created or deleted
  - There is no ‘heap’ or use of the C `malloc()` or `free()` functions
  - The converse is a ‘dynamic’ system, which is the opposite of all the above

- **Benefits of static systems:**
  - Reduced code size – create functions are replaced by BIOS declarations, there is no delete function, no inclusion of malloc/free functions or heap management
  - Reduced MIPS consumption for environment creation – no time spent in create/delete, `malloc()`/`free()`, etc
  - Deterministic performance – `malloc()` is non-deterministic
  - Optimal when most resources are required concurrently

- **Limitations of static systems:**
  - Fixed allocation of memory usage
  - Unable to create new components or modify existing ones at runtime

- **Bottom Line:**
  - Some systems are best served with a static configuration, others may benefit from a dynamic solution
  - DSP/BIOS fully supports both methodologies, even allowing easy migration between the two

### Static System Configuration Management

- **Segments** – memory blocks present in the target hardware
  - Properties: base address, length, usage (code/data)
  - *How are these defined in CCS?*

<table>
<thead>
<tr>
<th>Hardware</th>
<th>EPROM</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sections** – software component blocks, including:
  - System sections – eg: stack
  - C sections – eg: code,
  - BIOS sections – eg: code, objects, ...
  - User-defined sections - ex. #pragma CODE_SECTION
  - *How are these routed to the desired memory segment?*
GCONF & TCONF – Which One to Use?

- **GCONF (pre BIOS 5.x)** – Graphical Configuration Tool
  + TI's historical system setup method – created a .CDB file
  + Easy to use visual system design / wizard-like interface
  - Projects don’t directly transfer to new board, ISA, BIOS rev

- **TCONF (BIOS 5.0 – 5.1)** – Text-based Configuration
  + Java Script language; Write in any desired text editor
  + Produces smaller, more concise output
  + Output is printable – better documenting
  + Designed for easy transport to new board, ISA, BIOS revs
  + Runs under Linux, Solaris, Windows
  - User’s guide: SPRU007

- **GCONF (BIOS 5.2 +)** – Graphical to Text Config Tool
  + Set up a system as per GCONF
  + See TCONF components generated as you go
  + Outputs a TCONF resultant

- **CDB2TCF – CDB to TCF conversion utility or CDBCMP**
  + Converts existing GCONF .CDB files to TCONF .TCF format
  + Provided in BIOS 5.0 and greater
Hardware Segments

Defining Memory Segments

- Identify all hardware segments to CCS
- Each discontinuity requires a new segment
- A continuous block may be defined as separate sub-blocks if desired

1. right click on MEM mgr
2. select "insert MEM"
3. type MEM name
4. right click on new MEM
5. select "properties"
6. indicate desired
   - Base Address
   - Length
   - Space (Code, Data, both)
   - Heap: usage described later – leave this box unchecked in static systems

Note: Suppression of heap support is via: MEM manager | Properties | General | No Dynamic Memory Heaps

TCONF Memory Segment Setup

```javascript
/* load platform */
utils.loadPlatform("ti.platforms.dsk5510");

var myMem = MEM.create("IRAM");
myMem.comment = "internal RAM";
myMem.base = 0x00000000;
myMem.len = 0x00100000;
myMem.space = "both";
myMem.createHeap = "false";
/* myMem.heapSize = 0x08000; */
/* myMem.enableHeapLabel = "false"; */
/* myMem.heapLabel = prog.extern("seg_name", "asm"); */
MEM.NOMEMORYHEAPS = "true"; /* disable heap support */
```

Typical Memory Segments

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory Segment Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPRAM</td>
<td>Internal program memory</td>
</tr>
<tr>
<td>IDRAM</td>
<td>Internal data memory</td>
</tr>
<tr>
<td>SBSRAM</td>
<td>External SBSRAM on CE0</td>
</tr>
<tr>
<td>SDRAM0</td>
<td>External SDRAM on CE2</td>
</tr>
<tr>
<td>SDRAM1</td>
<td>External SDRAM on CE3</td>
</tr>
</tbody>
</table>
Software Sections

System Stack

DSP System Stack

- Stack is used for
  - Local variables in HWIs and SWIs (not TSKs)
  - Calling arguments for functions, Context save

- Configuration tool sets stack size and location at link time
  - Size: MEM.STACKSIZE or MEM -> Properties -> General -> Stack Size
  - Location: MEM.STACKSEG or MEM -> Properties -> BIOS Data -> Stack Section

- Create a large stack
- Fill with a known (non-zero) value
- Run system to exercise all likely usage
- Halt system, look for key value
  - No key values found? Increase stack size
  - Lots of key values left? Decrease stack size

- Alternate method
  - Use HWI monitor option on highest HWI(s)
  - Monitor SP
  - Look at max value(s) of SP

Managing Stack Size via GCONF, TCONF

- Minimum estimated stack size is shown at top of GCONF display.
- Stack size is set via:
  Memory Section Manager | Properties | General

MEM.STACKSIZE = 0x0400;
MEM.ARGSSIZE = 0x0004;
C Sections

C Program Sections

Global Variables
- `.bss`, `.far`
- Initial values
  - `.cinit`

Local Variables
- `.stack`

Dynamic Variables
- `.sysmem`
- Uninitialized sections
  - `.bss`
  - `.far`
  - `.stack`
  - `.sysmem`
  - `.cio`
- Initialized sections
  - `.text`
  - `.code`
  - `.switch`
  - `.const`
  - `.cinit`
  - `.pin`

Dynamic Variables
- malloc memory draw (heap)
- Buffers for stdio functions

Routing C Sections via GCONF

C components are routed to desired destinations via Memory Section Manager | Properties | Compiler Sections

MEM - Memory Section Manager Properties

General | BIOS Data | BIOS Code

- User and File For Non-DSP/BIOS Sections
  - Text Section (.text)
  - Switch Jump Tables (switch)
  - C Variables Section (.bss)
  - C Variables Section (.far)
  - Data Initialization Section (.cinit)
  - C Function Initialization Table (.pin)
  - Constant Section (.const)
  - Data Section (.data)
  - Data Section (.bss)

Load Addresses

IRAM | IIRAM

C components are routed to desired destinations via Memory Section Manager | Properties | Compiler Sections
## Routing C Sections via TCONF

MEM - Memory Section Manager Properties

<table>
<thead>
<tr>
<th>General</th>
<th>BIOS Data</th>
<th>BIOS Code</th>
<th>Compiler Sections</th>
<th>Load Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **User card file for Non-DSP/BIOS sections**

- **Text Section (.text):** IRAM
- **Switch Jump Tables (.switch):** IRAM
- **C Variables Section (.bss):** IRAM
- **C Variables Section (.data):** IRAM
- **Data Initialization Section (.init):** IRAM
- **C Function Initialization Table (.init):** IRAM
- **Constant Section (.const):** IRAM
- **Data Section (.data):** IRAM
- **Data Section (.bss):** IRAM
- **Data Section (.cso):** IRAM

- **MEM.TEXTSEG** = prog.get("IRAM");
- **MEM.SWITCHSEG** = prog.get("IRAM");
- **MEM.BSSSEG** = prog.get("IRAM");
- **MEM.FARSEG** = prog.get("IRAM");
- **MEM.CINITSEG** = prog.get("IRAM");
- **MEM.PINITSEG** = prog.get("IRAM");
- **MEM.CONSTSEG** = prog.get("IRAM");
- **MEM.DATASEG** = prog.get("IRAM");
- **MEM.CIOSEG** = prog.get("IRAM");
- **MEM.MALLOCSEG** = prog.get("IRAM");
BIOS Sections

BIOS Code

Hardware
- ROM
- EPROM
- SRAM
- Data
- DSP

Objects
- BUFF
- LOG
- STS
- Init
- Code

Functions

Routing BIOS Sections via GCONF

BIOS components are routed to desired destinations via Memory Section Manager.
Routing BIOS Sections via TCONF

MEM.BIOSSEG = prog.get("IRAM");
MEM.SYSINITSEG = prog.get("IRAM");
MEM.HWSEG = prog.get("IRAM");
MEM.HWVECSEG = prog.get("VECS");
MEM.RTDXTEXTSEG = prog.get("IRAM");

MEM.ARGSSEG = prog.get("IRAM");
MEM.STACKSEG = prog.get("IRAM");
MEM.GBLINITSEG = prog.get("IRAM");
MEM.TRCDATASEG = prog.get("IRAM");
MEM.SYSDATASEG = prog.get("IRAM");
MEM.OBJSEG = prog.get("IRAM");

Vector Setup via GCONF

Hardware Interrupt Vectors are routed via Memory Section Manager | Properties | BIOS Code | .hwi_vec to a segment defining the Vector range

Note: new seed files do not use separate VECS segment; .hwi_vec is first to link in IRAM
User Sections

User Defined Sections

- Use the memory segments created by the Configuration Tool
- Require a separate linker command file to place sections
- Place user defined sections using the SECTIONS directive

```c
#pragma DATA_SECTION(x, "mysect");
Int x[1024];
```

```cmd
SECTIONS
{
  mysect :> IRAM
}
```

- Use CCS “Link Order” capability to link the Config. Tool generated file then the user linker command file
- Put the user defined linker command file in the project (along with the BIOS command file)

Link Order

- Link Order is the fourth tab under Build Options
- The .cmd files are read in the order listed
- audiocfg.cmd is the Config. Tool’s .cmd file
  - MEM segments are declared here
- `user.cmd` must be added to the project
Files Created

Files Generated by the Config Tool

- **myWork.tcf**: Textual configuration script file
- **myWorkcfg.cmd**: Linker command file
- **myWorkcfg_c.c**: C file to s/u BIOS obj's, etc
- **myWorkcfg.s##**: ASM init file for series ## DSP
- **myWorkcfg.h##**: Header file for above
- **myWork.cdb**: I/F to GCONF display
- **myWorkcfg.h**: header file for config inclusions

File Extensions

- `audio.h`
- `audio.c`
- `audio.asm`
- `audio.tcf` (audio.cdb)
- `audio.cmd` (optional)
- `mod.h`
- Compiler/Assembler
- `audiocfg.obj`
- Linker
- `audio.out`
- `audio.obj`
- `audio.asm`
- Linker
- `audio.obj`
- `*.lib`
Observe & Tune Results

Post-Build Memory Usage Examination

- sectti.exe filename.out
  - Displays length and starting address of program sections in hex of COFF

- filename.map
  - Report generated by linker and specified in the build options

- ofd6x.exe
  - Object File Display (version for each ISA)
  - Provides a more detailed XML report than map file

System Tuning

- General Principles
  - Keep it simple: optimize locally, design globally
  - Make maximum use of internal memory – internal access is faster and lower power than external
  - Use Cache, DMA, where possible – multiply the effective size of internal resources

- Program Memory Optimization
  - Reduce memory size: CCS optimization options: -o3, -ms(0-2) ms “model size” – higher # biases optimization for size vs speed
  - Enable program caching – minimizing external code reads
  - Page in code sections using DMA – overlay next code over prior code while CPU reads current code: Management intensive!

- Data Memory Optimization
  - Reduce memory demand: use smaller arrays, unions
  - Enable data caching – minimize external reads and writes
  - Page in data channels via DMA – CPU services active channel, DMA concurrently imports next active channel
Startup Sequence

- Initialize the DSP and the hardware
  - The software stack pointer, memory wait states, memory configuration registers
  - This is part of the boot.c file that is part of the DSP/BIOS library
- BIOS_init() is called automatically
  - Initializes DSP/BIOS modules
- main()
  - System initialization that needs to be performed
  - Enable selected interrupts before interrupts are enabled globally
  - *Must return to complete the program initialization!!!*
- BIOS_start() is called automatically
  - Start DSP/BIOS
  - Enables interrupts globally
- Drops into the DSP/BIOS “background loop”
  - Initializes communication with the host for real-time analysis
Lab 9: Static System Management

A. Map File Inspection
1. Open CCS and load project myWork.pjt
2. Verify under the Linker tab of Project | Build Options that a map file was specified (if not, specify a map file and rebuild to obtain a map file)
3. Add the .map file into documents folder in the project tree. Open and inspect the map file. Near the top of the file note the amount of IRAM and DDR2 used. In the .far section, note the location and size of the memory allocated to proc.obj: __________________________
4. Expand the Memory Section Manager in the .TCF file. Note how the section names from the .map file correspond to those in the TCF file
5. Click on the Memory Section Manager and note that the properties shown in the right pane correspond to the results documented in the .map file

B. Remapping the Output Buffers
1. In proc.c, add a data section pragma to redefine the output buffer as being of type mysect
2. Create and add to the project a new file, myLink.cmd, which routes mysect to DDR2
3. Build, load, and run the program. Compare performance to the prior build (quality, load).
4. Inspect the new .map file with Windows Notepad. As before, note the usage of IRAM and DDR2 and the locations of the stream buffers __________________________
5. Save the results as per prior labs

C. Remapping the Input Buffers
1. Repeat the above steps to change the input stream buffers to DDR2
2. Again, build, load and run; note changes to the .map file. __________________________
3. Is the audio still as expected? Why? __________________________

D. Alignment of Buffers for Cache
The IOM manages the 6437 data cache when buffers are external. So, 128 byte buffer alignment and filling is required. Also, the history data must be saved in a local buffer before returning an old buffer back to the IOM, since the IOM will invalidate the data before it could be directly copied to the new buffer. A new version of proc.c will be used to demonstrate these constructs.
1. Copy the contents of C:\BIOS\Sols\09d to C:\BIOS\Labs\Work
2. Load workspace Lab09c.wks. Build, load, and run the program. Compare performance to the prior build (quality, load)
3. Inspect the new .map file to verify that the stream buffers are still off-chip.
4. Inspect proc.c to see the changes as described above
Dynamic Systems

Introduction

In this chapter the ability to create and delete system components will be considered as a way to reduce system size, make better use of on-chip resources, and tailor system components in response to events occurring as the system runs.

Objectives

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

- Contrast static and dynamic system coding benefits
- Implement dynamic BIOS object creation and deletion
- Contrast the BIOS MEM API to malloc and free
- Contrast MEM API with BUF API
- Modify static systems to employ dynamic methods

Module Topics

Dynamic Systems.................................................................................................................................10-1
  Concepts........................................................................................................................................10-2
  Dynamic Memory Allocation: MEM...............................................................................................10-3
  Dynamic Objects............................................................................................................................10-6
  Buffer Pools: BUF..........................................................................................................................10-8
  Review..........................................................................................................................................10-11

Lab 10: Dynamic Systems...........................................................................................................10-12
  The BIOS Kernal/Object Viewer...................................................................................................10-12
  A. Stream Buffers Obtained From Buffer Pool RAM .................................................................10-13
  B. Returning Buffers Back to the Pool : BUF_free()...............................................................10-14
  C. Heap-Based Buffers: MEM_alloc() & MEM_free()............................................................10-15
  D. Dynamic Task Creation ..........................................................................................................10-16
### Concepts

#### Static vs Dynamic Systems

<table>
<thead>
<tr>
<th>STATIC Environment</th>
<th>DYNAMIC Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Link-time</strong></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td>Declare streams</td>
<td>• Create streams</td>
</tr>
<tr>
<td>Declare buffers</td>
<td>• Allocate buffers</td>
</tr>
<tr>
<td></td>
<td>• Initialize variables</td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td><strong>Execute</strong></td>
</tr>
<tr>
<td></td>
<td>• Read data</td>
</tr>
<tr>
<td></td>
<td>• Process data</td>
</tr>
<tr>
<td></td>
<td>• Write data</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td><strong>Delete</strong></td>
</tr>
<tr>
<td></td>
<td>• Delete streams</td>
</tr>
<tr>
<td></td>
<td>• Free buffers</td>
</tr>
</tbody>
</table>

- Static Objects – created at link time via TCONF and/or GCONF
- Dynamic Objects – created at runtime via MOD_create() API
- Once created, both can be used identically
- Only dynamic objects may be deleted via MOD_delete() API
- Static benefits: easy, smaller code size, faster startup
- Dynamic benefits: smaller RAM budget, reuse of on-chip RAM
- Dynamic objects allow needs of the running system to determine the style and quantity of objects in place
Dynamic Memory Allocation: MEM

Dynamic Memory Allocation

```c
Ptr addr = MEM_alloc(Int segid, Uns size, Uns align);
```

- Superset of `malloc()` function – invokes the DSP/BIOS memory manager
- Allows selection of heap to draw from plus address alignment
- Creation of multiple heaps is via GCONF or TCONF
- Size is in NMADUs and allocation is always an even numbers of words
- Aligns on even boundaries
- Returns MEM_ILLEGAL if failure
- `malloc(size)` API is translated to `MEM_alloc(0,size,0)` in BIOS

```c
Ptr addr = MEM_calloc(Int segid, Uns size, Uns align);
```

- Is `MEM_alloc()` + clears (zeros) the array

```c
Ptr addr = MEM_valloc(Int segid, Uns size, Uns align, Char value);
```

- Is `MEM_alloc()` + fills array with specified value

```c
Bool status = MEM_free(Int segid, Ptr address, Uns size);
```

- Replacement for `free()` C function
- Complement to `MEM_alloc / valloc / calloc` APIs
- Must specify segment and size in addition to ptr to array
- Removal of auto-store of size arg allows better aligned array packing

Memory Heap Configuration

In *.c file, obtain access to heap label via:

```c
extern Int internal;
```
Memory Status Interrogation

```c
struct MEM_Stat {
    Uns size;    // original size of heap
    Uns used;    // number of MADUs used in heap
    Uns length;  // largest contiguous block available
}
```

```c
status = MEM_stat(segid, statbuf);
```

- Used to query the status of specified heap
- Helpful for diagnostics
- Could be used to actively manage heap usage in sophisticated system
- Cannot be called from a SWI or HWI; TSK scheduler must be enabled
- Same information available in CCS Kernel/Object View window

Dynamic Memory Considerations...

- **Non-Reentrant**
  - Cannot be called from within HWI or SWI

- **Non-Deterministic:**
  - Memory manager traverses linked list of free blocks for each allocate and delete
  - Recommended: Allocate and free memory during a background process or startup (pre-realtime)

- **Fragmentation:**
  - After repeated allocation and freeing of memory large contiguous blocks of memory may not be available to allocate
  - To minimize memory fragmentation:
    - Allocate smaller, equal-sized blocks of memory from one memory segment and larger equal-sized blocks of memory from a second segment, etc.

- *For all the above concerns: Consider using BUF API*
# MEM API Review

<table>
<thead>
<tr>
<th>MEM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM_alloc</td>
<td>Allocate memory from specified heap</td>
</tr>
<tr>
<td>MEM_calloc</td>
<td>MEM_alloc + clear (zero) the array</td>
</tr>
<tr>
<td>MEM_valloc</td>
<td>MEM_alloc + fill array with specified value</td>
</tr>
<tr>
<td>MEM_free</td>
<td>Return MEM_alloc’d array to specified heap</td>
</tr>
<tr>
<td>MEM_stat</td>
<td>Return the status of a memory segment</td>
</tr>
<tr>
<td>MEM_define</td>
<td>Define a new memory segment</td>
</tr>
<tr>
<td>MEM_redefine</td>
<td>Redefine an existing memory segment</td>
</tr>
</tbody>
</table>

```
  segid = MEM_define(base, length, attrs);
  MEM_redefine(segid, base, length);
```
Dynamic Objects

Dynamically Creating DSP/BIOS Objects

- **XXX_create**
  - Allocates memory for object out of segment defined in config tool
  - Returns a XXX_Handle to the created object

- **XXX_delete**
  - Frees the object’s memory
  - Argument: object handle; return type: void

```
#define COUNT 0
#include <sem.h>

SEM_Handle hMySem;
hMySem = SEM_create(COUNT,NULL);
SEM_post(hMySem);
SEM_delete(hMySem);
```

Modules supporting dynamic create and delete

<table>
<thead>
<tr>
<th>SWI</th>
<th>TSK</th>
<th>SIO</th>
<th>SEM</th>
<th>BUF</th>
<th>MSGQ</th>
<th>QUE</th>
<th>MBX</th>
<th>LCK</th>
<th>GIO</th>
</tr>
</thead>
</table>

Dynamic Object Creation and Deletion

```
hSwi   = SWI_create(attrs);  // attrs shown below
hTask  = TSK_create(fxn, attrs, [arg, ...]);
hStrm  = SIO_create(name, mode, bufsize, attrs);
hSem   = SEM_create(count, attrs);
hBuf   = BUF_create(numbuff, size, align, attrs);
hQueue = QUE_create(attrs);
hMbx   = MBX_create(msgsize, mbxlength, attrs);  // see below
hLock  = LCK_create(attrs);
hGio   = GIO_create(name,mode,*status,chanParams,*attrs)
```

- MOD_create API all return the handle to the new object
- Arguments mirror those from static configuration
- Many ‘attrs’ arguments are ‘placeholders’ for possible future definition
- **SEM, BUF, SIO, TSK examples appear elsewhere in this chapter**

```
struct SWI_Attrs {
  SWI_Fxn fnx;
  Arg arg0;
  Arg arg1;
  Int priority;
  Uns mailbox;
};

struct MBX_Attrs {
  Int segid;
  // heap to malloc from
};

struct MBX_Attrs {
  MBX_Attrs *attrs;  // pointer to mailbox attributes
  Uns msgsize;
  // size of message
  Uns mbxlength;
  // length of mailbox
};
```
**Dynamic Stream API**

```c
#include <sio.h>
#define BUFSIZE 100
SIO_Handle hStrmIn, hStrmOut;
SIO_Attrs attrs = SIO_ATTRS;
attrs.nbufs = 3;
hStrmIn = SIO_create("/a2d", SIO_INPUT, BUFSIZE, &attrs);
hStrmOut = SIO_create("/d2a", SIO_OUTPUT, BUFSIZE, &attrs);

// use In and Out streams as desired ...
SIO_idle(hStrmIn);
SIO_idle(hStrmOut);
SIO_delete(hStrmIn);
SIO_delete(hStrmOut);
```

**Dynamic Task API**

```c
#include <tsk.h>
TSK_Handle hMyTsk;
TSK_Attrs attrs = TSK_ATTRS;
attrs.priority = 3;
hMyTsk = TSK_create((Fxnn)myCode, attrs);

// "MyTsk" is now active in system with priority = 3 ...
TSK_delete(hMyTsk);
```
**Buffer Pools: BUF**

**BUF Concepts**

- **Buffer pools** contain a specified number of equal size buffers
- Any number of pools can be created
- Creation can be static (via TCONF) or dynamic (BUF_create / BUF_delete)
- Buffers are **allocated** from a pool and freed back when no longer needed
- Buffers can be **shared** between threads
- Buffer pool API are faster and smaller than malloc-type operations
- In addition, BUF_alloc and BUF_free are deterministic (unlike malloc)
- BUF API have no reentrancy or fragmentation issues
- Tip – larger buffers can be semi-filled by threads with smaller data blocks
- Cache and EDMA can cause coherency issues when using BUF or MEM API

---

**BUF_alloc() and BUF_free()**

- **pMyBuf = BUF_alloc(hPool)**
  - get a buffer from buffer pool `hPool` and initializes `pMyBuf` with the base address of the borrowed buffer
- **bStatus = BUF_free(hPool, pMyBuf)**
  - returns the buffer pointed to by `pMyBuf` to pool `hPool`
- Complement to BUF_alloc
- example of use:

```c
extern BUF_Obj bufferPool;
BUF_Handle hPool = &bufferPool;
Ptr pMyBuf;

pMyBuf = BUF_alloc(hPool);
if (pMyBuf == NULL )
    { SYS_abort("BUF_alloc failed");}

// thread can use buffer freely now...

bStatus = BUF_free(hPool, pMyBuf);
if(bStatus==0) {
    LOG_printf(&trace,"BUF_delete failed!");
}
```
**BUF_maxbuff() and BUF_stat()**

- `uCount = BUF_maxbuff(hPool);`
  - Returns maximum number of buffers in use at any time
  - Useful for fundamental system tuning and diagnostics
- `BUF_stat(hPool, pBufInfo);`
  - Information from the pool object is copied to the BufInfo structure
  - Useful for additional system tuning and diagnostics
- **example of use:**

```c
typedef struct BUF_Stat {
    MEM_sizep postalignsize;  // Size after align
    MEM_sizep size;           // Original size of buffer
    Uns totalbuffers;        // Total # of buffers in pool
    Uns freebuffers;         // # of free buffers in pool
} BUF_Stat;

BUF_Stat bufinfo;
extern BUF_Obj bufferPool;
BUF_Handle hPool = &bufferPool;
BUF_stat(hPool, &bufinfo);
LOG_printf(&trace, "Free buffers Available: %d", bufinfo.freebuffers);
```

---

**GCONF Creation of Buffer Pool**

**Creating a BUF**

1. right click on BUF mgr
2. select “insert BUF”
3. type BUF name
4. right click on new BUF
5. select “properties”
6. indicate desired
   - Memory segment
   - Number of buffers
   - Size of buffers
   - Alignment of buffers
   - Gray boxes indicate effective pool and buffer sizes
TCONF Creation of Buffer Pool

TCONF programming essentially mirrors the BUF_create arguments:

```c
myPool = BUF_create(numbuf, size, align, attrs);
```

- `BUF.OBJMEMSEG = prog.get("IRAM");` // mem segment for all pool objects
- `var myPool = BUF.create("myBuf");` // mem segment for all pool objects
- `myPool.bufCount = 8;` // number of buffers in myPool pool
- `myPool.size = 512*sizeof(short);` // size of each myPool buffer in MADUs
- `myPool.align = 2;` // alignment applied to each buffer in myPool
- `myPool.bufSeg = prog.get("IRAM");` // mem segment for 'myPool' buffers

```c
typedef struct BUF_Attrs {
    Int   segid;
} BUF_Attrs;
```

**BUF_create() and BUF_delete()**

- `hPool = BUF_create(numbuf, size, align, attrs)`
  - Dynamically create a buffer pool
  - Arguments: number of buffers in pool, their size and alignment, heap to draw from
  - Returns handle to pool object
  - Calls MEM_alloc() to obtain memory from heap
- `bStatus = BUF_delete(hPool)`
  - Frees a dynamically created buffer pool back to heap
  - Complement to BUF_create
  - Status: 1 = success, 0 = fail
  - Calls MEM_free to implement
- **example of use:**

```c
BUFF_Handle hPool;
BUFF_Attrs *myAttrs;
myAttrs = &BUF_ATTRS;

hPool = BUF_create(8, 1024, 2, &myAttrs);
if( hPool == NULL ) {LOG_printf(&trace,"BUF_create failed!");}
// buffer pool can now be used as long as desired...

bStatus = BUF_delete(hPool);
if( bStatus == 0 ) {LOG_printf(&trace,"BUF_delete failed!");}
```
BUF API Review

<table>
<thead>
<tr>
<th>BUF API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF_alloc</td>
<td>Allocate a buffer from the pool</td>
</tr>
<tr>
<td>BUF_free</td>
<td>Return a buffer to the pool</td>
</tr>
<tr>
<td>BUF_create</td>
<td>Dynamically create a pool of buffers</td>
</tr>
<tr>
<td>BUF_delete</td>
<td>Delete a dynamically created buffer pool</td>
</tr>
<tr>
<td>BUF_maxbuff</td>
<td>Interrogate maximum # buffers taken from pool</td>
</tr>
<tr>
<td>BUF_stat</td>
<td>Get pool info (buf size, # free bufs, total # bufs in pool)</td>
</tr>
</tbody>
</table>

Dynamic Systems - Review

- MOD_create() and MOD_delete()
  - Objects created as/when desired
  - Identical to static objects when executed
  - May be deleted when no longer required
  - Reduced RAM requirement
  - Larger code space, slower startup
  - System can adapt ‘on the fly’
  - MOD_create() returns a handle to the new object
  - MOD_delete() API are void return with handle arg

- MEM API
  - Allow dynamic array create and delete
  - Superset of malloc() and free()
  - Able to select from multiple heaps + alignment
  - Offers the most flexibility and memory savings ability
  - Issues: determinism, reentrancy, fragmentation

- BUF API
  - Allows sharing of buffers amongst threads
  - Buffers within a given pool are equal sized
  - Deterministic, reentrant, nonfragmenting
Lab 10: Dynamic Systems

In this lab the static declarations from lab 8 will be modified to use dynamic invocations. When RAM is at a premium, this may be an important way to get more use from finite resources.

The procedures for this lab do not provide details covered in earlier labs. Refer to prior labs or ask for assistance if there are difficulties in implementing the steps in the procedures that follow.

As noted, the start files needed for this lab will be the solution from the prior lab. If needed, a copy of this solution can be found in C:\BIOS\Sols\08e. If this particular chapter is of lesser interest, or if there is a time constraint, you may instead load the solution to this lab, also under the Sols directory, in the various subdirectories 10a through 10d, allowing you to skip over the authoring steps and move directly to seeing how the completed code looks and works.

The BIOS Kernal/Object Viewer

Inspect the actions of the dynamic API in each of the labs below via the Kernal Object Viewer, invoked via DSP/BIOS | Kernal/Object View…

In each lab, set breakpoints in the create, execute, and delete phases of the code. In order to improve the update speed of the KOV, check only the object types relevant to the given experiment. Click on the desired KOV element to see the details of that resource. Run to each breakpoint and note how the KOV provides metrics on the given resource state. For those labs where switch 3 caused the program to enter/exit various states, be sure to exercise this switch to observe this effect as well.

In the course of your observations with the KOV, you should have had an opportunity to observe the usage of heap in the IRAM, creation of SIO and TSK object, and the borrow/return of resources to the buffer pool.

It should also be noted that the KOV updates slowly, and care should be taken when single stepping or running to breakpoints to verify that the code halt has completed before invoking further progress. At best, updates to the KOV would have been missed; at worst, relaunching the code before a breakpoint has competed can lock up the debug session and require restarting CCS.
A. Stream Buffers Obtained From Buffer Pool RAM

1. Start CCS. Load the solution from lab8e. Verify the code builds and runs correctly

   In myWork.tcf:

2. Add a BUF object; Set properties to 5 buffers of length 1200 in lRAM

   In proc.c function procBuf:

3. In order to have pointers for two input and two output buffers, add two new pointers to short arrays pIn2 and pOut2 as task locals

4. Replace global arrays in and out with BUF_alloc() calls in the prolog of task procBuf

5. Modify the four SIO_issue initializations to use the pointers to the BUFS, rather than the prior references to the addresses of the declared buffers

6. Since there are no fixed buffers, the initialization of pPriorIn will no longer be valid, so declare it with no initial value. Instead, it can be set equal to the address of the 2nd allocated input buffer

7. Test the code: Build, download, and run. Verify the operation of the system using stream buffers obtained via BUF_alloc(). Debug if required
B. Returning Buffers Back to the Pool: 

In systems that employ buffer pools, it is often desirable to add the code at the end of the task to free the allocated buffers back to the pool when they are no longer required. In such cases, a complete task with create, execute, and delete phases will have been written. Follow the steps below to implement the delete phase to the results of the prior build.

In proc.c function procBuf:

1. **C/X/D banner comments**: As a visual convenience, add a banner comment just before the first BUF_alloc API indicating this is the ‘create phase’. Add another after the final allocation indicating the beginning of the “execute phase”. Finally, after the close brace of the while loop, add one indicating the start of the “delete phase”

2. **Create an exit condition**: Modify while(1) to while(!sw3). To reach the delete phase, the while loop test, previously a fixed “1” will now test DIP switch 3. Starting with the switch in the up position, the code would run as usual. When the switch is moved to the down postion, the condition fails, and the delete phase ensues

3. **Access to sw3**: Add a declaration of extern short sw3 to the list of external variables

4. **Idle the streams**: Begin the delete phase with SIO_idle and two SIO_reclaim()s for both streams. Without reclaiming the buffers, their ownership would not be back with the task

5. **Free the buffers allocated in the create phase**: Refer to the code from the create phase as a checklist for the BUF_free() API to be written in the delete phase

**Testing the code:

6. **Before testing**: Make sure that DIP switch 3 is in the UP position

7. **Test the code**: Build, download, and run. Verify normal operation of the audio system

8. **Test the delete phase**: Set a breakpoint at the first line of the delete phase. Press switch 3 down. Verify that execution has halted at the breakpoint. Use F10 to step through the remaining code. Where does control pass to when the TSK returns? Note: any new TSK that might be invoked from this point forward could re-use the buffer pool resources that this TSK had used previously. As such, multi-threaded systems can be readily authored to share resources, thus requiring less total memory
C. Heap-Based Buffers: \texttt{MEM\_alloc()} & \texttt{MEM\_free()}

\textit{In myWork.tcf:}

1. \textit{Verify the heap in IRAM:} Select \textbf{System | MEM | IRAM | Properties} and verify that the check box \textit{“create a heap”} is set and that the heap size is 0x8000.

2. Also verify that in \textbf{System | MEM | Properties | General} the Segment for \texttt{malloc()/free()} is \textit{IRAM}

\textit{In proc.c function procBuf:}

3. \textbf{Replace} the \texttt{BUF\_alloc()} API with \texttt{MEM\_alloc()} calls in the prolog of the TSK procBuf. For calling arguments use:
   - segID: \texttt{IRAM}
   - buffer size: (Rcv) 4*BUF+2*HIST (Xmt) 4*BUF
   - alignment : 2, or sizeof(short)

4. \textit{Declare IRAM memory section name as an external variable:} IRAM is a section label generated in the myWork.tcf BIOS script file. Currently the extern declaration for this variable is not included in myWorkcfg.h, so you will be required to add your own declaration in the global variables section. You will need to declare \texttt{extern Int IRAM;}

5. \textit{Free the buffers in the delete phase:} Replace the \texttt{BUF\_free()} calls with \texttt{MEM\_free()} API

6. \textit{Delete the stream:} Since the stream was created dynamically it too can be freed. After the buffers are reclaimed, use \texttt{SIO\_delete()} to free the stream object memory and also the hardware peripherals that were bound to the stream via \texttt{SIO\_create()}

7. \textit{Before testing:} Make sure that \textbf{DIP switch 3} is in the \textit{UP} position.

8. \textit{Test the code:} \textbf{Build, download, and run.} Verify the operation of the system using dynamically created stream buffers. Debug if required. Where do the buffers allocated reside? In what way are the buffers given out? Consult the .map file to determine how the heap location was selected

9. \textit{Test the delete phase:} Set a \textbf{breakpoint} at the first line of the \textit{delete phase}. \textbf{Press switch 3 down.} Verify that execution has halted at the breakpoint. Use \texttt{F10} to step through the remaining code. Where does control pass to when the TSK returns? \textit{Note: any new TSK that might be invoked from this point forward could re-use the heap resources that held the buffers this TSK had used previously. As such, multi-threaded systems can be readily be authored to share resources, thus requiring less total memory.}

10. \textit{Note:} if testing in release mode, setting a \textbf{breakpoint} in the \textit{delete phase} code may prove difficult. If so, simply run the program and halt to observe the resultant destination of the program counter.
D. Dynamic Task Creation

In this lab, a given solution will be provided to demonstrate the technique for dynamic invocation of BIOS objects with the use of TSK_create() on the procBuf TSK.

Load.c will be removed from the project and replaced with a new file which will use the same function calls and global variables but instead of implementing an artificial load, this new file will dynamically create and delete tskProcBuf when switch 3 is toggled.

1. Remove Load.c and NopLoop.asm from the project
2. Add dynamic_task.c from the Algos directory to the project. Now, DIP switch 3 will be used to create and delete the procBuf TSK. The switches no longer affect the (eliminated) load fxn
3. Examine dynamic_task.c
   - For ease of assimilation into the project, the task function name as tskLoad was not changed, even though this name does not reflect the new functionality
   - DIP switch 3 is read using EVMDM6437_I2C_read() from the board support library and compares this to the previous value (hw_sw3) to see if there has been a change
   - For a transition from low to high a new task is created, for high to low, the task is deleted
   - The task delete code waits for tskProcBuf to be reported as terminated before deleting it, ensuring that the cleanup that tskProcBuf does in its delete phase is allowed to complete
4. In myWork.tcf: Delete the statically defined tskProcBuf object
5. Make sure DIP switch 3 is in the Down (audio task ‘off’) position to start
7. Open the message log window and view the logDipSw log
8. Toggle dip switch 3. When it is in the up position, you should hear music, indicating that the process audio task has been created and is running. When sw3 is in the down position, the music will stop, indicating that the task has been deleted. Correspondingly, you will see messages in the message log indicating that the tskProcessBuffer task is being created and deleted with each switch toggle
9. Set breakpoints at the start of the create and delete phases of procBuf. Step through the code and observe the changes to the heaps and task in the KOV.
10. If desired, save the results to the mySols archive
11. What happens if the program is started with sw3 in the up position initially?

Here, in this lab, the benefit of the dynamic creation of this task is rather small as no other thread will acquire the memory it released back to the heap, and the environment of this task is simple and small. However, in larger system with complex tasks possessing significant sized environments, the benefit of dynamic task management could be substantial. Though the task object itself is not large, the stack which is allocated may be (default size is 1KB, but could be much larger). Hopefully, from these many lab exercises, it has been observed that systems can be developed statically, and once correct functionality has been achieved, dynamic management can be applied afterward with little difficulty.
Thread Comm and Sync

Introduction

Up to this point, the workshop has focused on the “IPO” (Input Process Output) topology common to most DSP systems. However, it is often required to be able to communicate between threads within a complex system. Previously, SEM (semaphores) were seen to be a way to signal between threads. In this chapter, SEM will be studied in greater depth, to see where it is optimal, and also where it can lead to unexpected complications. A number of other BIOS API are provided to improve on the inter-thread communication and synchronization options available to the user, and these will be shown here. Each will be overviewed for their use. The user should not how each compare and contrast so that the optimal technique can be selected for a given application.

Objectives

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

- Identify where semaphores are a good choice and when they can produce problems
- Define the atomic functions available in DSP/BIOS
- Explain how the LCK API can resolve some semaphore issues
- Demonstrate how to use mailbox (MBX) API to communicate between threads
- Describe how queues (QUE) are implemented in DSP/BIOS
- Demonstrate the use of Synchronous Communication (SCOM) extended BIOS API
- Describe the enhanced benefits of Message Queues (MSGQ)

Module Topics

Thread Comm and Sync ..........................................................................................................................11-1

  Concepts..............................................................................................................................................11-2
  Atomic Operations ..............................................................................................................................11-2
  Semaphores.......................................................................................................................................11-3
  LCK: Lock.........................................................................................................................................11-6
  MBX: Mailbox.....................................................................................................................................11-7
  QUE: Queues.....................................................................................................................................11-8
  MSGQ : Message Queue....................................................................................................................11-9
Critical Section Resource Protection

- Pass a copy of resources to 2nd thread
  - No possibility of conflict, allows both to use concurrently
  - Doubles storage requirements, adds copy time
- Assign concurrent threads to the same priority – FIFO
  + No possibility of conflict, no memory/time overhead, easy
  - Forces involved threads to be same priority
- Disable interrupts during critical sections
  + Needed if a hardware interrupt shares data
  - Affects response time of all threads in the system
- Disable SWI/TSK scheduler during critical section
  + Assures one SWI/TSK to finish with resource before another begins
  - Affects the response times of all other SWIs
- Raise priority of SWI/TSK during critical section
  * Set priority to highest priority which may access the resource
  + Equal priority SWIs/TSKs run in FIFO order, avoiding competition
  - Can affect response times of SWIs/TSKs of intervening priority
- Use atomic functions on shared resources
  + Imposes minimal jitter on interrupt latencies
  - Only allows minimal actions on shared resources
- Regulate access rights via Semaphores
  + No conflict or memory/time overhead of passing copy
  - Multiple semaphore schemes can introduce problems

Atomic Operations

DSP/BIOS Atomic Functions

- Allows thread to manipulate variables without interrupt intervention
- Are C callable functions optimized in assembly
- Allows reliable use of global variables shared between SWI and HWI

<table>
<thead>
<tr>
<th>TSK</th>
<th>x</th>
<th>ATM_dec</th>
<th>ATM_inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x+1;</td>
<td>10</td>
<td>- Good for inter-thread counters</td>
<td></td>
</tr>
<tr>
<td>LD reg, x</td>
<td>11?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD reg, 1</td>
<td>10 / 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST x, reg</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATM_clear</th>
<th>ATM_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Counters - to start/restart</td>
<td></td>
</tr>
<tr>
<td>- Generic pass of value between threads</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATM_and</th>
<th>ATM_or</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Perform boolean op’s on shared variables</td>
<td></td>
</tr>
<tr>
<td>- Good for event flags, etc</td>
<td></td>
</tr>
</tbody>
</table>
Semaphores

**Synchronization Semaphore**

*A and B same priority, A is ready before B*

- Priority=1
  - B
  - Priority=1
  - A

- SEM_pend(semObj)
- SEM_post(semObj)

**B higher priority than A, B is ready before A**

- Priority=2
  - B
  - Priority=1
  - A

- SEM_pend(semObj)
- Not dependent on A
  - block!
- Depends on A
  - SEM_post(semObj)
  - preempted!

**Semaphores and Priority**

- Priority=2
  - C
  - Priority=1
  - B
  - Priority=1
  - A

- interrupt!
- SEM_pend(&semObj)
  - block!
- Not dependent on A
- SEM_pend(&semObj)
  - block!
- Not dependent on A
- Depends on A
- SEM_post(&semObj)
  - preempted!
- Precondition for B and C

- Both B and C depend on A
- B pends on the semaphore first, then C
- When A posts, B runs first because it pended first
- Semaphores use a FIFO Queue for pending tasks!
Mutual Exclusion Semaphore

- Two or more tasks need concurrent access to a serial reusable resource
- Mutual exclusion using semaphore
  - Initialize semaphore count to 1
  - `pend` before accessing resource - lock out other task(s)
  - `post` after accessing resource - allow other task(s)

```c
Void task0()
{ SEM_pend(&semMutex);
  'critical section'
  SEM_post(&semMutex);
}

Void task1()
{ SEM_pend(&semMutex);
  'critical section'
  SEM_post(&semMutex);
}
```

- Problems may occur when tasks compete for more than one resource
  - Priority inversion
  - Deadlock

Priority Inversion

High-priority tasks block while waiting for lower-priority tasks to relinquish semaphore

```
Priority
higher

Initially
Mutex = 1
Pend(mutex)

Pend(mutex) blocks

post (mutex)

A

interrupt!

B

C

D

Post(mutex) preempted

preempted!

SEM_post(&semMutex)

SEM_pend(&semMutex)

Task0

Task1

Priority=2

Priority=1

interruption!

Time

"The failure turned out to be a case of priority inversion"

— Mars Pathfinder Flight Software Cognizant Engineer

void task0()
{ SEM_pend(&semMutex);
  'critical section'
  SEM_post(&semMutex);
}
void task1()
{ SEM_pend(&semMutex);
  'critical section'
  SEM_post(&semMutex);
}
Inversion Solution: Priority Inheritance

Question: Do we even need to use semaphore in this situation? No
Do not use TSK_yield() if semaphore is removed

Deadlock

- Also known as deadly embrace
- Tasks cannot complete because they have blocked each other
- TaskA and TaskB require the use of resource 1 and 2.
- Neither task will release a resource until it is completed

Conditions for deadlock to occur:
- Mutual exclusion: Access to shared resource protected with mutual exclusion SEM
- Circular pend: Circular chain of tasks hold resources that are needed by others in the chain (cyclic processing)
- Multiple pend and wait: Tasks lock more than one resource at a time
- Preemption: Tasks needing mutual exclusion are at a different priorities
**Deadlock: Detect, Recover, Eliminate**

- **Difficult to detect**
  - May happen infrequently
  - Use *timeouts* in blocking API; monitor timeout via SYS_error
  - Monitor SWI with implicit STS
- **Recover is not easy**
  - Reset the system
  - Rollback to a pre-deadlock state
- **Solution**
  - Careful design
  - Rigorous testing
- **Eliminating Deadlock: Remove one of these conditions**
  - Mutual exclusion: Make resources sharable
  - Circular pend: Set a particular order
  - Multiple pend and post: Lock only one resource at a time or all resources that will be used (starvation potential)
  - Preemption: Assign tasks that need mutual exclusivity to the same priority
- **Better**: Use more sophisticated BIOS API

---

**LCK: Lock**

**Nested Semaphore Calls: LCK**

Use of SEMaphore with nested pend **yields permanent block**

```c
Void Task_A()
{
    SEM_pend(&semUser);
    funcInner();
    SEM_post(&semUser);
}
```

Use of LCK (Lock) with nested pend **avoids blockout**

```c
Void Task_A()
{
    LCK_pend(&lckUser);
    funcInner();
    LCK_post(&lckUser);
}
```

Unrecoverable blocking call

Unrecoverable blocking call

---

Use of LCK (Lock) with nested pend **avoids blockout**

```c
Void Task_A()
{
    LCK_pend(&lckUser);
    funcInner();
    LCK_post(&lckUser);
}
```

Use of LCK (Lock) with nested pend **avoids blockout**

```c
Void Task_A()
{
    LCK_pend(&lckUser);
    funcInner();
    LCK_post(&lckUser);
}
```

BIOS MEM Manager and selected RTS functions internally use LCK, can cause TSK switch
MBX: Mailbox

Example: Passing Buffer Info Via Mailbox

```c
typedef struct MsgObj {
    Int len;
    Int * addr;
};

MBX_post - add message to end of mailbox
Void writer(Void)
{
    MsgObj msg;
    Int myBuf[SIZE];
    ...
    msg.addr = myBuf;
    msg.len = SIZE*sizeof(Int);
    MBX_post(&mbx, &msg, SYS_FOREVER);
    ...
}

MBX_pend - get next message from mailbox
Void reader(Void)
{
    MsgObj mail;
    Int size, *buf;
    ...
    MBX_pend(&mbx, &mail, SYS_FOREVER);
    buf = mail.addr;
    size = mail.len;
    ...
}
```

Mailbox benefits: message can be any desired structure, semaphore signaling built in (read and write), allows multiple readers and/or writers
Mailbox limitations: fixed depth of messaging, copy based – 2 copies made in/out of MBX

Creating Mailbox Objects

Message Object creation via GCONF

Message Object creation via TCONF

```
MBX.OBJMEMSEG = prog.get("ISRAM");
var myMBX = MBX.create("myMBX");
myMBX.comment = "my MBX";
myMBX.messageSize = 1;
myMBX.length = 1;
myMBX.elementSeg = prog.get("IRAM");
```

Dynamic Message Object Creation

```
hMbx = MBX_create(msgsize, mbxlen, attrs);
MBX_delete(hMbx);
```

Message Size = MADUs per message
Mailbox Length = max # messages queued
QUE: Queues

**Queues : QUE**

- QUE message is anything you like, starting with QUE_Elem
- QUE_Elem is a set of pointers that BIOS uses to manage a double linked list
- Items queued are **NOT** copied – only the QUE_Elem ptrs are managed!

```c
struct MyMessage {
    QUE_Elem elem;
    int x[1000];
} Message1;
```

```c
typedef struct QUE_Elem {
    struct QUE_Elem *next;
    struct  QUE_Elem *prev;
} QUE_Elem;
```

**QUE_put(hQue,*msg3)** add message to end of queue (writer)

```
QUE_Obj msg1 msg2 msg3
```

**QUE_get(hQue)** get message from front of queue (reader)

```
*elem = QUE_get(hQue)
msg1 QUE_Obj msg2 msg3
```

**How do you synchronize reader and writer?**

Queue benefits: any number of messages can be passed, message can be anything desired (beginning with QUE_elem), atomic API are provided to assure correct sequencing

Queue limitations: no semaphore signaling built in

**QUE API Summary**

<table>
<thead>
<tr>
<th>QUE API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE_put</td>
<td>Add a message to end of queue – atomic write</td>
</tr>
<tr>
<td>QUE_get</td>
<td>Get message from front of queue – atomic read</td>
</tr>
<tr>
<td>QUE_enqueue</td>
<td>Non-atomic QUE_put</td>
</tr>
<tr>
<td>QUE_dequeue</td>
<td>Non-atomic QUE_get</td>
</tr>
<tr>
<td>QUE_head</td>
<td>Returns ptr to head of queue (no de-queue performed)</td>
</tr>
<tr>
<td>QUE_empty</td>
<td>Returns TRUE if queue has no messages</td>
</tr>
<tr>
<td>QUE_next</td>
<td>Returns next element in queue</td>
</tr>
<tr>
<td>QUE_prev</td>
<td>Returns previous element in queue</td>
</tr>
<tr>
<td>QUE_insert</td>
<td>Inserts element into queue in front of specified element</td>
</tr>
<tr>
<td>QUE_remove</td>
<td>Removes specified element from queue</td>
</tr>
<tr>
<td>QUE_new</td>
<td>...</td>
</tr>
<tr>
<td>QUE_create</td>
<td>Create a queue</td>
</tr>
<tr>
<td>QUE_delete</td>
<td>Delete a queue</td>
</tr>
</tbody>
</table>
MSGQ Concepts (1/4)

- MSGQ transactions begin with listener opening a MSGQ
- Listener’s attempt to get a message results in a block (when semaphore specified), since no messages are in the queue yet

```c
MSGQ_open("que2", &q2, ...);
MSGQ_get( q2, &msg, ...);
```

MSGQ Concepts (2/4)

- Talker begins by locating the MSGQ opened by the listener
- Talker gets a message block from a pool and fills it as desired
- Talker puts the message into the MSGQ

```c
MSGQ_locate("que2", &q2, ..);
MSGQ_alloc( poolId, &msg, ..);
msg->myMsg = ...;
MSGQ_put( msg, q2 );
```
MSGQ Concepts (3/4)

- Once talker puts message to MSGQ, listener is unblocked
- Listener can now read/evaluate received message
- Listener frees message back to pool

MSGQ Concepts (4/4)

- The message object manages queuing of messages passed
- An allocator mechanism is for getting buffers; standard = POOL
- A transport mechanism can be specified for trans-processor MSGQ
Multiprocessor MSGQ

- MSGQ_locate doesn't find "que2" locally
- MSGQ_Transport (MQT) finds que2 on Proc1
- MSGQ_put sends block to MQT
- MQT sends data over physical link it manages
- Free of buffer back to local pool is implemented by MQT
- Listener TSK has no knowledge of location of talker
- Thread code is unchanged from local processor solution!
- SRIO versions provided by TI

MSGQ API

<table>
<thead>
<tr>
<th>writer</th>
<th>any</th>
<th>reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGQ_locate</td>
<td>MSGQ_open</td>
<td>once per object (either *)</td>
</tr>
<tr>
<td>MSGQ_allocate</td>
<td>MSGQ_put</td>
<td>ongoing...</td>
</tr>
<tr>
<td>MSGQ_release</td>
<td>MSGQ_get</td>
<td>once (or never) per object</td>
</tr>
<tr>
<td></td>
<td>MSGQ_free</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MSGQ_close</td>
<td></td>
</tr>
</tbody>
</table>

* it is recommended that MSGQ_open be a reader function – better suits multi-writer option
MSGQ Features:

+ any number of messages can be passed
+ message can be anything desired, beginning with MSGQ_header
+ semaphore signalling built in, messages passed by pointer
+ can be used between all thread types with no API adaptation
+ API unchanged even when going trans-processor!
Introduction

In this chapter the concepts and coding techniques of BIOS drivers will be considered. The original BIOS driver model “DEV” required the author develop all the code needed to interface with both the stream and the port hardware. The new ‘mini-driver’ or IOM standard allows the programmer to focus only on the port coding; the stream interface portion being managed by a TI-authored layer, thus simplifying the process of driver authoring. In addition the IOM, or MD (mini-driver), model offers greater flexibility and reuse, for once the MD is authored, it can be applied to the stream (SIO) model using the DIO interface, or the PIP model using a matching “PIO” interface. In addition, a generic interface (GIO) is defined for users who require a data transmission model beyond the limits of SIO or PIP. One such example is “FVID” (frame video), developed by TI for use in high-throughput video systems. Finally, it will be seen that the IOM model alleviates one additional difficulty in the original DEV model: the ability to author a devices that can support multiple streams. This simplifies coding of multi-channel IO ports such as serial ports, which support both transmit and receive ability. As shall be seen in this chapter, input and output streams can easily be connected to a single MD. This chapter presents a substantial amount of example code as a means of more fully demonstrating the concepts involved in authoring BIOS drivers.

Objectives

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

• Describe the concepts of BIOS drivers
• List the key MD API
• List the basic activities in each MD function
• Describe the support tools for writing IOMs
• Describe all components of an example IOM

Module Topics

IOM - I/O Mini-Drivers .................................................................................................................................12-1

IOM Concepts ...............................................................................................................................................12-2
DDK – Driver Developer’s Kit .....................................................................................................................12-4
CSL – Chip Support Library .......................................................................................................................12-7
Driver Structures .......................................................................................................................................12-11

MD Coding Example.................................................................................................................................12-14
mdBindDev...............................................................................................................................................12-15
mdCreateChan, mdDeleteChan ..................................................................................................................12-17
mdSubmitChan, mdControlChan ...............................................................................................................12-18
ISR Functions ...........................................................................................................................................12-21
Header File ...............................................................................................................................................12-22
IOM Concepts

SIO Review

- Common I/O interface: between Tasks and Devices
  - Universal interface to I/O devices
  - Yields improved code maintenance and portability
  - Number of buffers and buffer size are user selectable
- Unidirectional: streams are input or output - not both
- Efficiency: block passed by reference instead of by copy
  - _SIO_issue_ passes a “IOM_Packet” buffer descriptor to driver via stream
  - _SIO_reclaim_ waits for a IOM_Packet to be returned by driver via stream
- Abstraction: TSK author insulated from underlying functionality
  - BIOS (SIO) implicitly manages two QUEues (todevice & fromdevice)
  - _SIO_reclaim_ synchronized via implicit BIOS (DIO) SEM
  - IOM_Packets (aka DEV_Frames) produced by SIO on stream creation
- Asynchronous: TSK and driver activity is independent, synch’d by buffer passes
- Buffers: Data buffers must be created - by config tool or TSK

BIOS I/O Models

- Original DEV coding now broken into two parts
  - **Class Driver** – provided by TI
  - **I/O Mini Driver** – I/F to HW (port/peripheral); from DDK, etc
- **Process Thread Author**: Choose the class and I/O Mini-drivers desired
  - Same IOM for any class driver – “write once, use many’
  - Change of Mini-Drivers : new driver, same processing thread & class driver
MiniDriver: Interface to TSK or SWI

myTsk()
- SIO_create
- MEM_alloc
- SIO_issue
- SIO_issue

while(1)
- SIO_reclaim
dsp...
- SIO_issue

SIO_idle
- SIO_reclaim
- SIO_reclaim
- MEM_free
- SIO_delete

createSwi()
- SIO_create
- MEM_alloc
- SIO_issue
- SIO_issue

executeSwi()
- SIO_reclaim
dsp...
- SIO_issue

deleteSwi()
- SIO_idle
- SIO_reclaim
- SIO_reclaim
- MEM_free
- SIO_delete

IOM Methods – Concepts

BIOS_init
- mdBindDev

main

myTsk()
- SIO_create
- MEM_alloc
- SIO_issue
- SIO_issue

while(1)
- SIO_reclaim
dsp...
- SIO_issue

SIO_idle
- SIO_reclaim
- SIO_reclaim
- MEM_free
- SIO_delete

mdBindDev(dgp, devid, dparams)
- initialize parameters
- acquire resources
- initialize hw
- plug ISRs
- create/initialize global data structure (devObj)

mdCreateChan()
- create QUE
- create chan object
- enable interrupts

mdSubmitChan()
- if no current IOP, begin using this one
- else QUE for later use

IOM: ISR
- fill buf with data
- when full, run cbFxn(arg, IOP)

Class Driver: Provided by DSP/BIOS

DSK5402_MCBSP_AD50_init - usually an empty fn

DIO:
- create DIO object
- create QUE
- call mdCreateChan

DIO:
- dequeue IOP
- pass IOP to IOM
- call mdSubmitChan

DIO: SEM_pend()
- queue IOP
- SEM_post()
- rtn to IOM

DIO: callback fxn
- queue IOP
- SEM_post()
DDK – Driver Developer’s Kit

Device Driver Developer's Kit: DDK

- Productized Drivers for TI DSP Peripherals
- Simple Example Applications that Use These Drivers
- Documentation on Using Existing Drivers & Developing New Drivers
- Downloadable: Free of Charge / No Run-time Royalties
- Available via:
  - CCS Update Advisor
  - TI DSP Developer's Village (dspvillage.com)
  - www.TI.com
- Example Drivers:

<table>
<thead>
<tr>
<th>Platform</th>
<th>McBSP</th>
<th>UART</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>6711 DSK</td>
<td>AD535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6713 DSK</td>
<td>AIC23</td>
<td>SW</td>
<td>McASP</td>
</tr>
<tr>
<td>6416 DSK</td>
<td>AIC23</td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>DM642 EVM</td>
<td>AIC23</td>
<td>HW</td>
<td>McASP</td>
</tr>
<tr>
<td>C6x1x</td>
<td>AD535</td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>C5416</td>
<td>PCM3002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5509</td>
<td>AIC23</td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>C5510</td>
<td>AIC23</td>
<td>SW</td>
<td></td>
</tr>
</tbody>
</table>

DDK Documentation

Start Here: DSP/BIOS Driver Developer's Guide (SPRU616)

<table>
<thead>
<tr>
<th>Doc #</th>
<th>DSP</th>
<th>Device</th>
<th>Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPRA882</td>
<td>All</td>
<td>UART</td>
<td>All</td>
</tr>
<tr>
<td>SPRA858</td>
<td>C5000</td>
<td>McBSP/DMA</td>
<td>All C5000</td>
</tr>
<tr>
<td>SPRA857</td>
<td>C5509</td>
<td>AIC23 Codec</td>
<td>C5509 DSK</td>
</tr>
<tr>
<td>SPRA855</td>
<td>C5416</td>
<td>PCM3002 Codec</td>
<td>C5416 DSK</td>
</tr>
<tr>
<td>SPRA856</td>
<td>C5510</td>
<td>AIC23 Codec</td>
<td>C5510 DSK</td>
</tr>
<tr>
<td>SPRA846</td>
<td>C6x1x</td>
<td>McBSP/EDMA</td>
<td>All C6000</td>
</tr>
<tr>
<td>SPRA850</td>
<td>C6x11</td>
<td>AD535 Codec</td>
<td>C6711 DSK</td>
</tr>
<tr>
<td>SPRA677</td>
<td>C6713</td>
<td>AIC23</td>
<td>C6713 DSK</td>
</tr>
<tr>
<td>SPRA909</td>
<td>C6416</td>
<td>AIC23</td>
<td>C6416 DSK</td>
</tr>
<tr>
<td>SPRA677</td>
<td>DM642</td>
<td>AIC23</td>
<td>DM642 EVM</td>
</tr>
<tr>
<td>SPRA870</td>
<td>C6x1x</td>
<td>McBSP/EDMA</td>
<td>All C6000</td>
</tr>
</tbody>
</table>

- Each Device Driver Has Corresponding Documentation (App Note)
  - Usage, Architecture, Data Sheet
- Every Device Driver project has a Readme File
DDK Top-Level Directory Structure

c:\ddk_1_20\packages\ti\bios\drivers
- c54xx_dma_mbbsp
  + - - doc
  + - - c5416
  + - - debug
  + - - release
  + - - c55xx_dma_mbbsp
  + - - c5509A
  + - - c5510
  + - - c6x1x_edma_mbbsp
  + - - c6713
  + - - cDM642
- c6x1x_edma_mbbsp
  + - - c6416
  + - - c6711
  + - - c6713
- dsk5416_dma_pcm3002
- dsk5510_dma_aic23
- dsk6416_edma_aic23
- dsk6713_edma_aic23
- dsk6x11_edma_ad535
- evm5509A_dma_aic23
- evmDM642_edma_aic23
- evmDM642_uarthw

DDK Top-Level Directory Structure

+ - - examples
  + - - audio
  + - - dsk5416
  + - - dsk5510
  + - - dsk6416
  + - - dsk6711
  + - - dsk6713
  + - - evm5509A
  + - - evmDM642
  + - - uart
  + - - dsk5510
  + - - dsk6416
  + - - dsk6713
  + - - evm5509A
  + - - evmDM642
+ - - pio
+ - - shared
+ - - uarthw
  + - - uarthw_c55xx_mbbsp
  + - - c5509A
  + - - c5510
  + - - uarthw_c6x1x_mbbsp
  + - - c6416
  + - - c6713
+ - - uartmd

- examples applications
- audio driver example source
- dsk5416 audio example
- uart driver examples source
- PIP to IOM class driver interface
- DDK shared header file
- Generic DSP drivers sources and headers for Uart
- C5000 specific source and headers
- chip specific uart driver
- C6000 specific source and headers
- chip specific uart driver
- chip specific uart driver
- chip specific uart driver
- Generic Software UART driver source & headers
DDK Summary

- Productized IOM Drivers for TI DSP Peripherals
  - PCI, USB, Multimedia Card, McBSP, McASP, Video Ports, Codecs, UART
  - Full source code and documentation provided
- Introduces the new IOM Driver Model to simplify development of new drivers
  - Reusable modules
  - Standard APIs defined
  - Backwards-compatibility is maintained for older BIOS code which uses older driver models
- Extensible, integrated DSP/BIOS I/O Modules
  - New DEV, PIO, GIO API’s

DSP/BIOS McBSP Codec Driver

- Generic McBSP-DMA Data Mover
  - Implemented as a stand-alone mini-driver
  - Multi-channel
  - Reusable across codecs
- Codec Specific Part of Mini-Driver
  - Handles codec specific bind, channel open
  - AIC23, PCM3002, AD50, AD535
- Only mdSubmitChan and mdCreateChan Calls Are Handled by the Codec-Specific Portion of the Mini-Driver, So That’s All You Have To Write!
CSL – Chip Support Library

Chip Support Library - CSL

- CSL is a collection of:
  - Functions - to create/delete and interact with peripherals
  - example: MCBSP_config() [Module_function]
  - Structures - that define the control registers of the peripheral
  - example: MCBSP_Config() [Module_Type]
  - Macros - that allow complex register fields to be directly managed
  - Symbols - well defined to improve readability/maintainability

- Goals of CSL:
  - Ease of use
  - Faster development time
  - Improved portability across TI DSPs
  - Enhanced readability / documentation of functionality
  - Hardware abstraction

- Benefits of CSL:
  - Standardized protocol
  - Basic resource management
  - Peripheral symbol definitions

Create:

```
mdBindDev() --> hMcBsp = MCBSP_open(channel, ...)
```

McBSP channel can be: 0, 1, 2, ‘any’

CSL manages availability of ‘registered’ peripherals via “ChipDefine”

```
CSL : BSP flags: 0 1 0
```

```
MCBSP_config(hMcBSP, ...)
MCBSP_start(hMcBSP, ...)
```

Execute:

```
SIO_issue() --> mdSubmitChan() -->
SIO_reclaim() --> ISR -->
MCBSP_read(hMcBSP)
MCBSP_write(hMcBSP, ...)
```

Delete:

```
SIO_delete() --> mdDeleteChan() -->
MCBSP_close(hMcBSP)
```
CSL Coding Example – EDMA Peripheral

```
// 1. include headers
#include <csl.h>
#include <csl_edma.h>

// 2. Initialize CSL
CSL_init();

// 3. make a handle
EDMA_Handle hMyChannel;

// 4. open a periph, get handle – periph 'checked out' by CSL
hMyChannel = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);

// 5. define config structure
EDMA_Config myConfig
{
    /*specify all register/bit values here...*/
}

// 6. configure the channel
EDMA_config (hMyChannel, &myConfig);
```
CSL Coding – Interrupt Management

// 1. include headers
#include <csl_irq.h>
// 2. enable specific interrupt
IRQ_enable(IRQ_EVT_EDMAINT);
// 3. enable interruptability
IRQ_globalEnable( );

// option – adjust configuration using Field Make (FMK) macro
myConfig.opt |= EDMA_FMK (reg, field, value);

/* Open DMA and MCBSP */
hDmaRx = DMA_open( DMA_CHAANY, DMA_OPEN_RESET );
hMcbsp = MCBSP_open( MCBSP_PORT0, MCBSP_OPEN_RESET );

/* Set up configuration for DMA and MCBSP */
DMA_config( hDmaRx, &dmaMcbspRx );
MCBSP_config( hMcbsp, &mcbspCfg0 );

/* Start the DMA and MCBSP */
DMA_start( hDmaRx );
MCBSP_start( hMcbsp, MCBSP_RCV_START );
Chip Support Library Topology

◆ Modular library:
  - DMA   ICACHE
  - IRQ   MCBSP
  - TIMER WDTIM
  - GPIO  DAT
  - PWR
  * any interdependencies automatically link in required components

◆ 5910 library:
  - small model - csl5910dsp.lib
  - large model - csl5910dspx.lib
  - device support symbol - CHIP_5910

◆ CSL Directories:
  - Library   c:\ti\c5500\bios\lib
  - Source    c:\ti\c5500\bios\lib
  - Headers   c:\ti\c5500\bios\lib\include
  - Examples  c:\ti\examples\target\csl
  - Documentation c:\ti\docs
Driver Structures

IOM Functions

typedef struct IOM_Fxns {
    IOM_TmdBindDev mdBindDev;
    IOM_TmdUnBindDev mdUnBindDev;
    IOM_TmdControlChan mdControlChan;
    IOM_TmdCreateChan mdCreateChan;
    IOM_TmdDeleteChan mdDeleteChan;
    IOM_TmdSubmitChan mdSubmitChan;
} IOM_Fxns;

initialize port on BIOS startup
initialize port on BIOS startup
currently null fxn, poss. future use
response to SIO.ctrl()
response to SIO.create()
response to SIO.delete()
SIO_issue, _idle, _abort response

Also, an ISR/HWI is required within the IOM to collect/output the data to/from the
buffer issued to the IOM, and route the completed buffer back to the processing
thread (TSK, SWI) as a response to the SIO_reclaim() API

IOM Packet Descriptor

typedef struct IOM_Packet {
    QUE_Elem link; // used by SIO to manage buffer queue
    Ptr addr; // address of buffer
    Uns size; // size of buffer
    Arg misc; // callback fxn for packet stored here
    Arg arg; // anything you like (usually nothing)
    Uns cmd; // for 'submit', action for IOM to perform on this packet
    Int status; // IOM writes return status of packet here
} IOM_Packet;

- IOM Packet descriptors are filled in by DSP/BIOS (in SIO and DIO) using:
  - properties defined via stream creation
  - arguments of SIO_issue
- IOM authors will reference IOM_Packet fields to know what it needs to do and how

in iom.h : typedef DEV_Frame IOM_Packet;
Example Channel Object

typedef struct ChanObj {
    Bool inuse; // know if channel currently 'open'
    Int mode; // options: IOM_INPUT or IOM_OUTPUT
    IOM_Packet *dataPacket; // packet (descriptor) of buffer currently in use
    QUE_Obj pendList; // queue of packets waiting to be used
    Uns *bufptr; // ptr to next element in buffer for service
    Uns bufcnt; // how many elements left to service
    IOM_TiomCallback cbFxn; // function to call when done with this buffer
    Ptr cbArg; // return status of buffer
} ChanObj, *ChanHandle;

- Driver and Channel Objects are defined by the IOM author (not a universal standard across all IOMs, like an IOM_Packet is)
- This example contains elements that most IOMs will likely require
- More sophisticated IOMs will likely require more extensive objects

Information Exchange Between Structures

status = SIO_issue(hStream, pBuf, uSize, arg);

IOM_Packet
QUE_Elem link;
Ptr addr;
Uns size;
Arg misc;
Arg arg;
Uns cmd;
Int status;

hChan → ChanObj

uSize = SIO_reclaim(hStream, *pBuf, pArg);
### IOM Status and Error Codes

<table>
<thead>
<tr>
<th>Status/Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOM_COMPLETED</td>
<td>0 Successful completion</td>
</tr>
<tr>
<td>IOM_PENDING</td>
<td>1 I/O queued &amp; pending</td>
</tr>
<tr>
<td>IOM_FLUSHED</td>
<td>2 Packet flushed</td>
</tr>
<tr>
<td>IOM_ABORTED</td>
<td>3 Packet aborted</td>
</tr>
<tr>
<td>IOM_EBADADDR</td>
<td>-1 Generic failure</td>
</tr>
<tr>
<td>IOM_EINUSE</td>
<td>-2 Timeout occurred</td>
</tr>
<tr>
<td>IOM_ENOPACKETS</td>
<td>-3 No packets available</td>
</tr>
<tr>
<td>IOM_EFREE</td>
<td>-4 Unable to free resources</td>
</tr>
<tr>
<td>IOM_EALLOC</td>
<td>-5 Unable to allocate resources</td>
</tr>
<tr>
<td>IOM_EABORT</td>
<td>-6 I/O aborted uncompleted</td>
</tr>
<tr>
<td>IOM_EBADMODE</td>
<td>-7 Illegal device mode</td>
</tr>
<tr>
<td>IOM_EOF</td>
<td>-8 End-of-file encountered</td>
</tr>
<tr>
<td>IOM_ENOTIMPL</td>
<td>-9 Operation not supported</td>
</tr>
<tr>
<td>IOM_EBADARGS</td>
<td>-10 Illegal arguments used</td>
</tr>
<tr>
<td>IOM_ETIMEOUTUNREC</td>
<td>-11 Unrecoverable timeout</td>
</tr>
<tr>
<td>IOM_ABORTED</td>
<td>-12 Device already in use</td>
</tr>
</tbody>
</table>

### Example Device Object

```c
typedef struct MyDevObj {
    ChanObj Input; // know if channel currently 'open'
    ChanObj Output; // options: IOM_INPUT or IOM_OUTPUT
    McBSP_Handle hMcBSP; // packet (descriptor) of buffer currently in use
    Bool curInit; // queue of packets waiting to be used
    DSK5402...Params PARAMS
} MyDevObj, *MyDevHandle;
```

- Driver Object is defined by the IOM author (not an IOM standard)
- This example is just a pair of Channel Objects
- Most IOMs will likely require more extensive objects
- The AD50 example *could* have used a Dev Object as follows:
MD Coding Example

Includes, Globals, Local Fxn Prototypes

```c
#include <std.h>
#include <atm.h>
#include <hwi.h>
#include <que.h>
#include <iom.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include <dsk5402_mcbsp_ad50.h>
#include <ad50.h>

static MCBSP_Handle hMcbsp; // CSL McBSP object handle
DSK5402_MCBSP_AD50_DevParams DSK5402_MCBSP_AD50_DEVPARAMS = { AD50_DEFAULTPARAMS }; // default AD50 properties
static Void rxIsr(void); // isr fxns – ‘heart’ of any IOM
static Void txIsr(void);
static Void updateChan(ChanHandle chan);
static Void abortio(ChanHandle chan);
```

Channel Object and Device Object

```c
typedef struct ChanObj {
  Bool inuse;
  Int mode;
  IOM_Packet *dataPacket;
  QUE_Obj pendList;
  Uns *bufptr;
  Uns bufcnt;
  IOM_TiomCallback cbFxn;
  Ptr cbArg;
} ChanObj, *ChanHandle;

#define INPUT  0 // input chan is chanObj 0
#define OUTPUT 1 // output channel obj is 2nd struc in chans array
#define NUMCHANS 2 // ‘chans’ is an array of 2 channel objects

static ChanObj chans[NUMCHANS] = {
  { FALSE, INPUT, NULL, { NULL, NULL },
    NULL, 0, NULL, NULL },
  { FALSE, OUTPUT, NULL, { NULL, NULL },
    NULL, 0, NULL, NULL }
};
```

Know if channel currently ‘open’
Options: IOM_INPUT or IOM_OUTPUT
Packet (descriptor) of buffer currently in use
Queue of packets waiting to be used
Ptr to next element in buffer for service
How many elements left to service
Function to call when done with this buffer
Return status of buffer

‘Chans’ is also the Device Object in this example
Both channels initially begin in null state
Two bracketed nulls are for queue head & tail ptrs
IOM Fxn Prototypes and vTab

```c
// ===== Forward declaration of IOM interface functions / Prototypes
static Int mdBindDev(Ptr *devp, Int devid, Ptr devParams);
static Int mdControlChan(Ptr chanp, Uns cmd, Ptr args);
static Int mdCreateChan(Ptr *chanp,  Ptr devp, String name, Int mode,
                      Ptr chanParams, IOM_TiomCallback cbFxns, Ptr cbArg);
static Int mdDeleteChan(Ptr chanp);
static Int mdSubmitChan(Ptr chanp, IOM_Packet *packet);

// =========== Public IOM interface table / vTab
IOM_Fxns DSK5402_MCBSP_AD50_FXNS = {
    mdBindDev,
    IOM_UNBINDDEVNOTIMPL,
    mdControlChan,
    mdCreateChan,
    mdDeleteChan,
    mdSubmitChan
};
```

- Most IOMs will provide a nearly identical copy of the above code
- The only likely difference will be the name of the FXNS table

**mdBindDev**

**md Handle Passing**

```
BIOS_init() → mdBindDev()

hDev "devp" ↓

SIO_create() → mdCreateChan()

hChan "chanp" ↓

SIO_issue() → mdSubmitChan()

*IOP
```
**mdBindDev()**

`status = mdBindDev(*devp, devid, devParams);`

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>*devp</td>
<td>return address for global device data pointer</td>
</tr>
<tr>
<td>Int</td>
<td>devid</td>
<td>device id - used if more than one instance of the driver is to be created</td>
</tr>
<tr>
<td>Ptr</td>
<td>devParams</td>
<td>pointer to config parameters</td>
</tr>
<tr>
<td>Int</td>
<td>Status</td>
<td>returns success/failure of function</td>
</tr>
</tbody>
</table>

```c
#pragma CODE_SECTION(mdBindDev, ".text:init")
static Int mdBindDev(Ptr *devp, Int devid, Ptr devParams)
{
  DSK5402_MCBSP_AD50_DevParams *params = (DSK5402_MCBSP_AD50_DevParams *)devParams;
  static Bool curinit = FALSE;
  static MCBSP_Config mcbspCfg0 = {
    0x0021, 0x0201, 0x0040, 0x0000, 0x0040, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x000c, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
  };
  static volatile ioport unsigned port04;
  if (curinit) { return (IOM_EBADIO); }
  if (params == NULL){ params = &DSK5402_MCBSP_AD50_DEVPARAMS; }
  curinit = TRUE;
  if (params == NULL){ params = &DSK5402_MCBSP_AD50_DEVPARAMS; }  
  hMcbsp = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
  MCBSP_config(hMcbsp, &mcbspCfg0);
  port04 &= 0xf5;
  MCBSP_start(hMcbsp, MCBSP_XMIT_START | MCBSP_RCV_START, 0x0);
  AD50_setParams(hMcbsp, &params->ad50);
  HWI_dispatchPlug(IRQ_EVT_RINT1, (Fxns)xIsr, NULL);
  HWI_dispatchPlug(IRQ_EVT_XINT1, (Fxns)xIsr, NULL);
  *devp = chans;
  return (IOM_COMPLETED);
}
```
**mdCreateChan, mdDeleteChan**

### mdCreateChan()

\[
\text{status} = \text{mdCreateChan}(*\text{chanp}, \text{devp}, \text{name}, \text{mode}, \text{chanParams}, \text{cbFxn}, \text{cbArg});
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*chanp</td>
<td>Pointer</td>
<td>Return address for channel handle</td>
</tr>
<tr>
<td>devp</td>
<td>Ptr</td>
<td>Handle to device (global data structure)</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Name of device or instance</td>
</tr>
<tr>
<td>mode</td>
<td>Int</td>
<td>Direction of data flow: INPUT or OUTPUT</td>
</tr>
<tr>
<td>chanParams</td>
<td>Ptr</td>
<td>Pointer to channel parameters</td>
</tr>
<tr>
<td>cbFxn</td>
<td>IOM_TiomCallback</td>
<td>Pointer to callback function</td>
</tr>
<tr>
<td>cbArg</td>
<td>Ptr</td>
<td>Pointer to callback function argument</td>
</tr>
</tbody>
</table>

### mdDeleteChan()

\[
\text{status} = \text{mdDeleteChan}(*\text{chanp})
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chanp</td>
<td>Ptr</td>
<td>Handle to channel</td>
</tr>
</tbody>
</table>

### mdCreateChan()

```c
static Int mdCreateChan(Ptr *chanp, Ptr devp, String name, Int mode,
Ptr chanParams, IOM_TiomCallback cbFxn, Ptr cbArg)
{
    ChanHandle chans = (ChanHandle)devp;
    ChanHandle chan;
    if (mode == IOM_INPUT) { chan = &chans[INPUT];} 
    else if (mode == IOM_OUTPUT) { chan = &chans[OUTPUT]; } 
    else { return (IOM_EBADMODE);}
    if (ATM_setu((Uns *)&chan->inuse, TRUE)) { return (IOM_EBADIO);}
    QUE_new(&chan->pendList);
    chan->dataPacket = NULL;
    chan->cbFxn = cbFxn;
    chan->cbArg = cbArg;
    if (chan->mode == INPUT) { IRQ_enable(IRQ_EVT_RINT1); }
    else { IRQ_enable(IRQ_EVT_XINT1); }
    *chanp = chan;
    return (IOM_COMPLETED);
}
```
mdDeleteChan()

```
static Int mdDeleteChan(Ptr chanp)
{
    ChanHandle chan = (ChanHandle)chanp;
    chan->inuse = FALSE;
    if (chan->mode == INPUT) {
        IRQ_disable(IRQ_EVT_RINT1);
    } else {
        IRQ_disable(IRQ_EVT_XINT1);
    }
    return (IOM_COMPLETED);
}
```

get channel ID from arguments
channel is no longer in use
if channel is input, then
disable receiver interrupts
otherwise, channel is output
and transmit interrupt is disabled
return indicating deleteChan completed

mdSubmitChan, mdControlChan

```
mdSubmitChan(), mdControlChan

status = mdSubmitChan (chanp, *packet);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>channp</td>
<td>Handle to channel</td>
</tr>
<tr>
<td>IOM_Packet</td>
<td>*packet</td>
<td>Descriptor of buffer sent to md</td>
</tr>
<tr>
<td>Int</td>
<td>Status</td>
<td>returns success/failure of function</td>
</tr>
</tbody>
</table>

```
status = mdControlChan (chanp, cmd, arg);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>channp</td>
<td>Handle to channel</td>
</tr>
<tr>
<td>Uns</td>
<td>cmd</td>
<td>Control function to perform</td>
</tr>
<tr>
<td>Ptr</td>
<td>Arg</td>
<td>Optional – for driver specific data structure</td>
</tr>
<tr>
<td>Int</td>
<td>Status</td>
<td>returns success/failure of function</td>
</tr>
</tbody>
</table>
mdSubmitChan()

```c
static Int mdSubmitChan(Ptr chanp, IOM_Packet *packet)
{
    ChanHandle chan = (ChanHandle)chanp;
    Uns imask;
    if (packet->cmd == IOM_FLUSH ||
        packet->cmd == IOM_ABORT){
        abortio(chan);
        packet->status = IOM_COMPLETED;
        return (IOM_COMPLETED);
    }
    imask = HWI_disable();
    if (chan->dataPacket == NULL) {
        chan->bufptr = (Uns *)packet->addr;
        chan->bufcnt = packet->size;
        chan->dataPacket = packet;
    } else {
        QUE_put(&chan->pendList, packet);
    }
    HWI_restore(imask);
    return (IOM_PENDING);
}
```

- Handle to ChanObj
- pGIE
- if API was SIO_flush or
  SIO_abort
- call the abort fxn (later slide)
- mark the packet 'done'
- return "OK"

- If API was SIO_issue, turn off ints
- if there is no current packet (buffer)
- point to the top of the new buffer
- reset the buffer counter
- set this packet as the 'current' packet

- if I already have a packet in progress
- put this one in the queue for later
- put GIE back to prior state
- return 'buffer in progress'

mdControlChan()

```c
static Int mdControlChan(Ptr chanp, Uns cmd, Ptr args)
{
    if (cmd == IOM_CHAN_TIMEDOUT) {
        abortio(chanp);
    } else {
        return (IOM_ENOTIMPL);
    }
    return (IOM_COMPLETED);
}
```

- IOM author can select any number of control operations desired
- In this example, only a handler for stream timout was implemented
- DIO manages the test for timeout. If timeout occurs, DIO calls mdControlChan
- Final option to return 'not implemented' should be present in all mdControlChan
  functions to respond to commands not supported by the IOM
mdControlChan() – “Volume” Example

```c
static Int mdControlChan(Ptr chanp, Uns cmd, Ptr args)
{
    if (cmd == IOM_CHAN_TIMEDOUT) {
        abortio(chanp);
    }
    if (cmd == MY_IOM_VOL) {
        chanp->vol = (short) args;
    } else {
        return (IOM_ENOTIMPL);
    }
    return (IOM_COMPLETED);
}
```

- Example: adding an IOM control option - suppose this IOM had a ‘volume control’ (eg: ranging scale on ADC)
- Add the volume parameter “vol” to the channel object
- Test for the VOL command – if present, adjust “vol” in the chanObj
- Refer to chanp.vol for ADC ranging option in IOM code as applicable
- Note in IOM documentation / .h file MY_IOM_VOL command, type and allowed range of args values (generally a structure, here a simple short)

abortio

```c
static Void abortio(ChanHandle chan)
{
    iOM_Packet *tmpPacket;
    HWI_disable();
    tmpPacket = chan->dataPacket;
    chan->dataPacket = NULL;
    HWI_enable();
    if (tmpPacket) {
        tmpPacket->status = IOM_ABORTED;
        (*chan->cbFxn)(chan->cbArg, tmpPacket);
    }
    tmpPacket = QUE_get(&chan->pendList);
    while (tmpPacket != (IOM_Packet *)&chan->pendList) {
        tmpPacket->status = IOM_ABORTED;
        (*chan->cbFxn)(chan->cbArg, tmpPacket);
    }
}
```

Called in response to:
- TSK: SIO_flush()
- TSK: SIO_abort()
- DIO: stream timeout

critical section – atomic mode
save a copy of the curr data pkt info now there is no active data pkt
as long as there are pkts
as above... mark the pkt ‘aborted’ run the cbFxnx to rtn the pkt
get another pkt frm the Q

ISR Functions

ISR functions

static Void rxIsr(Void)
{
    ChanHandle chan = &chans[INPUT];
    if (chan->dataPacket == NULL) {
        MCBSP_read(hMcbsp); // dummy read
        return;
    }
    *chan->bufptr = MCBSP_read(hMcbsp);
    updateChan(chan);
}

static Void txIsr(Void)
{
    ChanHandle chan = &chans[OUTPUT];
    if (chan->dataPacket == NULL) {
        MCBSP_write(hMcbsp, 0); // dummy write
        return;
    }
    MCBSP_write(hMcbsp, *chan->bufptr & 0xfffe);
    updateChan(chan);
}

ISR subroutine: updateChan

static Void updateChan(ChanHandle chan)
{
    IOM_Packet *tmpPacket;
    chan->bufptr++;
    chan->bufcnt--;
    if (chan->bufcnt == 0) {
        chan->dataPacket->status = IOM_COMPLETED;
        tmpPacket = chan->dataPacket;
        chan->dataPacket = QUE_get(&chan->pendList);
        if(chan->dataPacket == (IOM_Packet *)&chan->pendList)
            chan->dataPacket = NULL;
        else {
            chan->bufptr = chan->dataPacket->addr;
            chan->bufcnt = chan->dataPacket->size;
        }
        (*chan->cbFxn)(chan->cbArg, tmpPacket);
    }
}
#ifndef DSK5402_MCBSP_AD50_
#define DSK5402_MCBSP_AD50_
#include <iom.h>
#include <ad50.h>
/* Driver function table to be used by applications. */
extern IOM_Fxns DSK5402_MCBSP_AD50_FXNS;
/* Setup structure for the driver (contains only codec registers) */
typedef struct DSK5402_MCBSP_AD50_DeVParams {
    AD50_Params ad50;           /* codec parameters (registers) */
} DSK5402_MCBSP_AD50_DeVParams;
/* Name of the default device params structure, defined in the driver module */
extern DSK5402_MCBSP_AD50_DeVParams DSK5402_MCBSP_AD50_DEVPARAMS;
/* Mini-driver init function -- initializes driver variables, if any */
extern Void DSK5402_MCBSP_AD50_init( Void);
#endif

#pragma CODE_SECTION(DSK5402_MCBSP_AD50_init, ".text:init")
Void DSK5402_MCBSP_AD50_init(Void) { }

init function from .c file – as is often the case, it is an empty function...
Introduction

In this chapter the DSP Algorithm Standard will be investigated. The value of using XDAIS algorithms in a complex system will be considered, and the method to adapt a given algorithm to the XDAIS standard will be explored.

Objectives

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

- Describe the benefits of using XDAIS compliant algorithms
- Describe how users control algorithm behaviour
- Describe how users control algorithm’s RAM and ROM usage
- Describe how XDAIS algorithms support multiple instances
- List the interface methods required in XDAIS algorithms
- List the sequence of actions in using a XDAIS algorithm
- Use Component Wizard to develop an XDAIS interface

Module Topics

- XDAIS Concepts
- XDAIS Benefits
- XDAIS Details
  - Instance Creation Parameters
  - Data Memory Management
  - Program Memory Management
  - Multi Instance Ability
  - XDAIS = Static or Dynamic
  - XDAIS Chronological Overview
  - XDM: XDAIS For Digital Media
- XDAIS Coding
- XDAIS Code Review
- DMA and XDAIS
  - EDMA Hardware Overview
  - XDAIS DMA Resource Management
  - ACPIY3 API
  - iDMA API
  - DMAN3 Framework
DSP Software Development Challenges

Problem
- Ever increasing code size in DSP systems
- Pressure to reduce time to market
- Pressure to reduce cost of development
- Too much for any one person – or team

Solution
- Distributed development
- Parallel authoring speeds time to market
- Work can be divided by expertise
- Third party ‘off the shelf’ components

Methodology
- Divide software into “system integration” and “component authoring” levels
- Allows for maximum reuse of components
- Allows for system specific usage by integrators

Applications:
- Consumer
- Commercial
- Portable
- etc

Components...
- Drivers
- Algorithms

Distributed Development Concerns

Problem:
Getting all the disparately written pieces to ‘fit together’

Solution: Develop to a universal standard
- Everyone knows what to expect
- Learn once, use repeatedly
- Reuse: author to standard and pieces will ‘fit’
- Makes use of 3P SW a practical reality
Software Standard Concerns

**Problem:**
- How to author such a standard?
- How to get others to apply it?
- How to make the standard maximally usable?
- How to minimize overhead when authoring to the standard?
- How to assure a component complies with the standard?

**Solution:**
- Already written – by TI
- 100’s of 3P’s already using it
- 1000’s of algos today, designed to be generic and non-limiting to algo
- Overhead can be as small as a few words and cycles
- TI provides testing software and offers compliancy validation svc

Available eXpressDSP Compliant Algos

<table>
<thead>
<tr>
<th>Audio</th>
<th>Speech</th>
<th>Motor Ctrl</th>
<th>Encryption</th>
<th>VB Modem</th>
<th>Video / Imaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Stereo</td>
<td>AAC Decoder/Encoder</td>
<td>Adaptive Noise Canceller</td>
<td>Adaptive Speech Filter</td>
<td>3-DES AES</td>
<td>ACG BELL 103, 202</td>
</tr>
<tr>
<td>AAC Decoder/Encoder</td>
<td>Acoustic</td>
<td>ASR Densifier/Noise Canceller</td>
<td>Assembly DES</td>
<td>V.21, V.22, V.23</td>
<td></td>
</tr>
<tr>
<td>Acoustic Echo Canceller</td>
<td>Adaptive</td>
<td>broadband Call ID Text to Speech</td>
<td>Diffie-Hellman</td>
<td>V.32, V.34, V.42, V.90</td>
<td></td>
</tr>
<tr>
<td>Adaptive Noise Canceller</td>
<td>Chorus Effect</td>
<td>Full Duplex/Noise Suppression</td>
<td>ELGAMAL</td>
<td>more . . .</td>
<td></td>
</tr>
<tr>
<td>Chorus Effect</td>
<td>MP3</td>
<td>MPEG4 Speech Decoder</td>
<td>HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>Decoder/Encoder</td>
<td>Voice Recognition</td>
<td>MD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>MPEG2</td>
<td>Voice Recognition</td>
<td>RSA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>MPEG4</td>
<td>Voice Recognition</td>
<td>more . . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>Noise Reduction</td>
<td>Voice Recognition</td>
<td>Wireless</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>Reverb</td>
<td>Voice Recognition</td>
<td>2.28 bps/Hz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>PTCM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Cyclic Redundancy Check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Deinterleaver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Multiplexer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Viterbi Decoder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>more . . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Telephony</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>2100Hz Tone Dec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Call Progress Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Caller ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>DTMF Echo Canceler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Line Echo Canceler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>Tone Detector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect</td>
<td>more . . .</td>
<td>Voice Recognition</td>
<td>more . . .</td>
<td></td>
<td></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
- TMS320 DSP Algorithm Standard Developer’s Kit Available to help develop your own compliant IP
Application / Component Advantages

Dividing software between components and system integration provides optimal reuse partitioning, allowing:

- **System Integrator:** full control of system resources
- **Algorithm Author:** to write components that can be used in any kind of system

What are “system resources”?...

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() )
- SI can give algo memory permanently (static systems) via declaration or just during life of algo (a “dynamic system”) via malloc-like action
- 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

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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>Scratch A</td>
<td>Per.A</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>Scratch B</td>
<td>Per.B</td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scratch B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Per.B</td>
</tr>
</tbody>
</table>

Okay for speed-optimized systems, but not where minimum memory usage is desired ...

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scratch</th>
<th>Per.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>Scratch A</td>
<td>Per.A</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>Scratch B</td>
<td>Per.B</td>
</tr>
<tr>
<td>Algorithm C</td>
<td>Scratch C</td>
<td>Per.C</td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Per.B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Per.C</td>
</tr>
</tbody>
</table>

Usually a: Limited Resource  e.g.: Internal RAM
Often an: Extensive Resource  e.g.: External RAM
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
  \(\text{algActivate()}, \text{algDeactivate()}\)
- 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

Example of Benefit of Scratch Memory

<table>
<thead>
<tr>
<th># Chans</th>
<th>Std</th>
<th>w. 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>10</td>
<td>10,000</td>
<td>1320</td>
</tr>
</tbody>
</table>

1K Block

FIR size = 32

Workspace = blocksize + firsize

History = firsize
Resource Management: ROM / Code

- **System Integrator controls:**
  - Which algos reside in which ROM
  - Which functions reside in which ROM
  - Which functions are linked in or discarded
- Normally, with a single ROM resource, everything goes into the same place – nothing fancy
- When multiple ROM resources are present, SI enjoys the ability to decide which algos/fxns go to a given ROM – better resource control
- In simple systems (eg: static systems), dynamic create/delete fxns may be of no interest to the SI, and may be discarded so as to not waste space with unused fxns
- **Tactical constructs involved:** vTab, code space pragmas, linker directives

---

2 vs 3 Layer Topology

Algo can be directly managed by Application, ie: a '2 layer' topology
- Allows most direct interface between system and component
- Requires more coding effort since algo IIF is at a low level
- All management and details must be managed by SI/App author
- But, if all algos work identically, couldn't this management be proceduralized to an intermediary level/layer?

SI can employ a 'framework' to reduce app code's algo management to higher level constructs, eg: Create, Execute, Delete

- Frameworks can be independently written to manage algos in whatever manner the SI desires (static v dynamic, scratch mgmt policies, etc)
- TI has written several frameworks already, provided free to TI DSP users
  - Source code is provided: allowing code to be observed, tweaked, reused
  - Already written, tested, documented: save time, high reliability, easy to use
- DaVinci Codec Engine includes two such frameworks:
  - "DSKT2" to manage memory
  - "DMAN3" to manage DMA resources (channels, PaRAMs, TCCs)
### DSKT2 Module

**Initialization Phase (config-time)**
- **SRAM:**
  - 0x8000_0000-0x8010_0000
- **IRAM:**
  - 0x0000_0000-0x0004_0000

**Usage Phase (run-time)**
- **Alg1:** 20K SRAM, 5K IRAM
- **Alg2:** 10K SRAM, 10K IRAM

- Acts as a warehouse for Memory resources
- System integrator initializes the DSKT2 module with available memory resources
- Algorithms “check out” memory from the DSKT2 module at runtime when they are instantiated.

---

### DSKT2 – Scratch Memory Sharing

- **Algorithm 1**
  - Persistent Memory #1
  - Shared Scratch Memory

- **Algorithm 2**
  - Persistent Memory #2

- Additionally, DSKT2 assigns scratch memory to algorithms for sharing of valuable internal memory.
- Scratch memory may be used when memory values do not need to be maintained between algorithm process calls
- The system integrator must provide a scheme for guaranteeing that algorithms which share scratch memory do not pre-empt each other. (next section)
DSKT2 Framework

- Designed to manage all memory needs of all xDAIS algs in system
- Presents a simple “Create, Execute, Delete” interface to application level
- Dynamic framework – persistent memories are only allocated when an algorithm is created, and are returned to heap when algo is deleted
- Scratch memory requests cause allocation of memory ‘pool’ that can be shared by other algs with the same “Scratch Pool ID”
  - Allows reuse/sharing of scratch pool RAM resource
  - Scratch pools only created when a user in that group has been created
  - Tip: assign same SGID to algs of matching priority
  - Tip: balance number of priorities vs number of scratch pools
  - Tip: Predefine default size of SG pools or create largest users first

<table>
<thead>
<tr>
<th>DSKT2 method</th>
<th>Sub operations performed…</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSKT2_createAlg</td>
<td>algNumAlloc algAlloc MEM_alloc algInit</td>
</tr>
<tr>
<td>DSKT2_activateAlg</td>
<td>algActivate</td>
</tr>
<tr>
<td>DSKT2_controlAlg</td>
<td>algControl</td>
</tr>
<tr>
<td>DSKT2_deactivateAlg</td>
<td>algDeactivate</td>
</tr>
<tr>
<td>DSKT2_freeAlg</td>
<td>algNumAlloc algFree MEM_free</td>
</tr>
</tbody>
</table>

- These API are managed by the Codec Engine in DaVinci systems
- They are presented here “FYI” for Codec Engine users
- Non Codec Engine systems can use these API directly to manage xDAIS algs

VISA – CODEC Engine – DSKT2 - xDM

VISA API Layer: User Interface

<table>
<thead>
<tr>
<th>VIDDEC_create()</th>
<th>VIDDEC_control()</th>
<th>VIDDEC_process()</th>
<th>VIDDEC_delete()</th>
</tr>
</thead>
<tbody>
<tr>
<td>algNumAlloc</td>
<td>control</td>
<td>process</td>
<td>algNumAlloc</td>
</tr>
<tr>
<td>algAlloc</td>
<td></td>
<td></td>
<td>algFree</td>
</tr>
<tr>
<td>MEM Alloc</td>
<td></td>
<td></td>
<td>MEM_free</td>
</tr>
<tr>
<td>algInit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CODEC engine / framework Layer: TI Infrastructure

<table>
<thead>
<tr>
<th>algNumAlloc</th>
<th>algAlloc</th>
<th>algInit</th>
<th>algFree</th>
</tr>
</thead>
<tbody>
<tr>
<td>algNumAlloc</td>
<td>algAlloc</td>
<td>algInit</td>
<td>algFree</td>
</tr>
<tr>
<td>MEM Alloc</td>
<td>MEM Alloc</td>
<td>MEM Alloc</td>
<td>MEM Alloc</td>
</tr>
<tr>
<td>algInit</td>
<td>algInit</td>
<td>algInit</td>
<td>algInit</td>
</tr>
<tr>
<td>algDeactivate</td>
<td>process</td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>algActivate</td>
<td>algDeactivate</td>
<td></td>
<td>control</td>
</tr>
<tr>
<td>algInit</td>
<td>algInit</td>
<td>algInit</td>
<td>algInit</td>
</tr>
<tr>
<td>algDeactivate</td>
<td>process</td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>algActivate</td>
<td>algDeactivate</td>
<td></td>
<td>control</td>
</tr>
</tbody>
</table>

xDM Algo Code Layer: DSP Algo Author Focus

- These API are managed by the Codec Engine in DaVinci systems
- They are presented here “FYI” for Codec Engine users
- Non Codec Engine systems can use these API directly to manage xDAIS algs
XDAIS Benefits

XDAIS Strategic Question: Make vs. Buy?

Tactical Requirements:
1. Static vs. Dynamic Algorithms
2. Memory Reuse Amongst Algorithms
3. Uniform Algorithm Interface
4. Improved Software Modularity
5. Multiple Instances of Algorithms
6. Improved Re-usability of Algorithms

Strategic Benefits:
1. Faster Time to Market
2. Reduced Software Development Costs
3. Improved Software Reliability
4. Ability to Leverage Special Expertise
5. “Off the Shelf” Convenience

Faster to Market?

Won’t I spend just as much – or more – time integrating a purchased algorithm into my system than just writing it myself?

Prior to the standard, integration effort could be quite substantial, as it often happened that new software would conflict with existing components.

However, the XDAIS rules were developed to assure problem-free component integration, eliminating this concern.
Reducing Software Development Costs...

Isn’t purchasing algorithms adding expense?

What are the present costs for:
  Authoring (engineering time + overhead)
  Debug, Verification
  Lost time to market (profit, reputation)

If analysing all the above indicates it’s better to make a given algorithm than buy it, then write your own – with XDAIS.

If other algorithms are better bought than made, then XDAIS allows their integration into your system with no complications stemming from their ‘foreign’ authorship…

Improved Software Reliability

How do I know purchased software will be reliable?

How do you know ‘home grown’ code is reliable? Apply the same testing to the purchased algo that you would your own. Given their standard interface, it’s easy to integrate a new XDAIS algo into your system, and you can quickly determine if the algo is worth having.

In fact, XDAIS algos add reliability in two ways.

First, since they’re so easy to integrate, several options can be tested and the best selected. Without XDAIS this is much harder to do, and there is usually no ‘apples to apples’ comparison between algos due to their differing interfaces and other design choices made.

Even more compelling is the fact that many algos will have already been in use in other systems, proving their reliability in practice more than any amount of lab testing might have shown on one’s own code.
### Ability to Leverage Special Expertise

Are all the aspects of every system you will build completely covered by in-house expertise? Is that expertise available when you need it?

Outside expertise can be ‘rented’ instead of ‘bought’ when needed for solving problems outside the expertise or interests of your company.

Why spend the time becoming an expert in a splinter topic if it slows your time to market? Instead, spend that time becoming better yet at what you do best. Trade the time being broad for the ability to achieve greater depth in the areas that differentiate your company.

### Ability to Select “Best in Class” per Algo...

- **Interfacing**
- **Performance**
- **Power**
- **Size**
- **Ease-of Use**
  - Programming
  - Interfacing
  - Debugging
- **Cost**
  - Device cost
  - System cost
  - Development cost
  - Time to market
- **Integration**
  - Memory
  - Peripherals
Instance Creation Parameters

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>Defined by</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated by</td>
<td>Application</td>
</tr>
<tr>
<td>Written to by</td>
<td>Application</td>
</tr>
<tr>
<td>Read from by</td>
<td>Algorithm</td>
</tr>
</tbody>
</table>

`sizeof()`

<table>
<thead>
<tr>
<th>Defined by</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated by</td>
<td>Application</td>
</tr>
<tr>
<td>Written to by</td>
<td>Application</td>
</tr>
<tr>
<td>Read from by</td>
<td>Algorithm</td>
</tr>
</tbody>
</table>

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;
```

```c
// 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,
};
```

- Defines Parameter Defaults
- Length of Structure
- Filter Length
- Block Size
- Coefficient Pointer

- 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;

IFIR_params.blockSize = 1000;
```

---

**Data Memory Management**

**The MemTab Transaction**

- **Algorithm**
  - Knows memory requirements
  - Requests appropriate resources from Application

- **Application (Framework / Node)**
  - Manages memory requests
  - Determines what memories are available to which algorithms - and when

- **Physical Memory “space”:**
  - External (slow, plentiful, less cost)
  - Internal (fast, limited, higher cost)
  - SARAM, DARAM

- **Params**
  - sizeOf
  - *coeffPtr
  - filterLen
  - frameLen

- **memTab**
  - size
  - alignment
  - space
  - attrs
  - address0
  - size
  - alignment
  - space
  - attrs
  - address1
  - size
  - alignment
  - ...
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>Defined by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>IALG Spec &amp; Algorithm (rtn value of algNumAlloc)</td>
</tr>
<tr>
<td>space</td>
<td>Allocated by:</td>
</tr>
<tr>
<td>attrs</td>
<td>Application</td>
</tr>
<tr>
<td>*base</td>
<td>5*algNumAlloc()</td>
</tr>
<tr>
<td></td>
<td>Written to by:</td>
</tr>
<tr>
<td></td>
<td>Algorithm (4/5) &amp; Application (base addr)</td>
</tr>
<tr>
<td></td>
<td>Read from by:</td>
</tr>
<tr>
<td></td>
<td>Algorithm (base addr)</td>
</tr>
</tbody>
</table>

Key Framework (eg DSKT2) Code

```
Determine number of buffers required
n = fxns->ialg.algNumAlloc();

Build the memTab
memTab = (IALG_MemRec *) MEM_alloc(0, n*sizeof(IALG_MemRec), 0);

Inquire buffer needs from algo
n = fxns->ialg.algAlloc((IALG_Params *)params,&fxnsPtr,memTab);

Allocate memory for algo
for (i = 0; i < n; i++)
    memTab[i].base = (Void *)MEM_alloc(memTab[i].space, memTab[i].size, memTab[i].alignment);

Set up handle, `fxns pointer, initialize instance object
alg = (IALG_Handle)memTab[0].base;
alg->fxns = &fxns->ialg;
fxns->ialg.algInit(alg, memTab, NULL, (IALG_Params *)params)

Free the memTab & return the handle to the newly created instance object
MEM_free(0, memTab, n*sizeof(IALG_MemRec));
return ((IFIR_Handle)alg);
```
The vTab Concept and Usage

```
#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;
```

```
hFir->fxns=&FIR_TTO_IFIR;
```

```
#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;
```
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 algo 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:algControl > SDRAM, type = NOLOAD
.text:algFree > SDRAM, type = NOLOAD
.text:algMoved > SDRAM, type = NOLOAD
.text:algNumAlloc > SDRAM, type = NOLOAD
.text > SDRAM
```
Multi Instance Ability

The Instance Object Structure

**Purpose:** To allow the application to specify to the algorithm the desired modes for any options the algorithm allows, e.g., size of arrays, length of buffers, Q of filter, etc…

- `*fxns`
- `filterLen`
- `blockSize`
- `*coeffs`
- `*workBuf`
- ...
- ...

Defined by: *Algorithm*
Allocated by: *Application*
Written to by: *Algorithm*
Read from by: *Algorithm*

(Private structure!)

Multiple Instances of an Algorithm

Allocate, Activate as many instances as desired
Uniquely named handles allow control of individual instances of the same algorithm

All instance objects point to the same vtab
Coefficient array can be shared
Scratch can be separate or common as desired
XDAIS = Static or Dynamic

Static vs. Dynamic Algorithms

<table>
<thead>
<tr>
<th>“Normal” (static) C Coding</th>
<th>“Dynamic” C Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define SIZE 32</td>
<td>#define SIZE 32</td>
</tr>
<tr>
<td>int x[SIZE]; /<em>allocate</em>/</td>
<td>Create</td>
</tr>
<tr>
<td>int a[SIZE];</td>
<td></td>
</tr>
<tr>
<td>x={…}; /<em>initialize</em>/</td>
<td></td>
</tr>
<tr>
<td>a={…};</td>
<td></td>
</tr>
<tr>
<td>filter(...); /<em>execute</em>/</td>
<td>Execute</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
</tr>
<tr>
<td></td>
<td>free(x);</td>
</tr>
<tr>
<td></td>
<td>free(a);</td>
</tr>
</tbody>
</table>

While DaVinci is a fully dynamic system, xDAIS was designed to support either static or dynamic systems.

Support for Static & Dynamic Instances

<table>
<thead>
<tr>
<th>Static *</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Framework</td>
</tr>
<tr>
<td>algInit</td>
<td>Create</td>
</tr>
<tr>
<td>Filter</td>
<td>Execute</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: Static case would also invoke “algActivate” if algo uses “scratch” memory
XDAIS Chronological Overview

Instance Creation - start

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

2. How many blocks of memory will you need to do this for me?

3. I’ll make a place where you can tell me about your memory needs...
   ```c
   *memTab = malloc(5*N)
   ```

Instance Creation - finish

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

5. Tell me about your memory requirements...

6. My needs, given these parameters, are this, for each of the N blocks of memory

7. I’ll go get the memory you need...
   ```c
   for(i=0;i<=N;i++)
   base=malloc(size);
   ```

8. Prepare an instance to run!

9. Copy Params and memory bases into my instance object…
### Instance Execution

<table>
<thead>
<tr>
<th>Application Framework</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Get ready to run. Scratch memory is yours now.</td>
<td><code>algActivate()</code></td>
</tr>
<tr>
<td>3. Run the algorithm …</td>
<td><code>runDSP()</code></td>
</tr>
<tr>
<td>5. I need the scratch block back from you now…</td>
<td><code>algDeactivate()</code></td>
</tr>
</tbody>
</table>

2. Prepare scratch memory, as required, from *persistent* memory

4. Perform algorithm - freely using all memory resources assigned to algo

6. Save scratch elements to persistent memory as desired

### Instance Deletion

<table>
<thead>
<tr>
<th>Application Framework</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If I no longer need the algorithm:</strong></td>
<td><strong>MemTab</strong></td>
</tr>
<tr>
<td>1. I’ll make a memTab again, or reuse the prior one</td>
<td><code>size</code>&lt;br&gt;<code>alignment</code>&lt;br&gt;<code>space</code>&lt;br&gt;<code>attrs</code>&lt;br&gt;<code>*base</code></td>
</tr>
<tr>
<td><code>*memTab = malloc(5*N)</code></td>
<td><code>InstObj</code></td>
</tr>
<tr>
<td>2. What memory resources were you assigned?</td>
<td>Param1&lt;br&gt;Param2&lt;br&gt;…&lt;br&gt;Base1&lt;br&gt;Base2&lt;br&gt;…</td>
</tr>
<tr>
<td>4. <em>free</em> all persistent memories recovered from algorithm</td>
<td>3. Re-fill memTab using <code>algAlloc</code> and base addresses stored in the instance object</td>
</tr>
</tbody>
</table>
**XDM: XDAIS For Digital Media**

**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**

If given a packaged algo (only object code), what would you need to know to use it?  

- **Need to Know**  
- **“Packaged” Algo**  
- **Filter Example**  

<table>
<thead>
<tr>
<th>What config options do I have?</th>
<th>Algorithm Parameters</th>
<th>“xDAIS” functions</th>
<th>my DSP functions</th>
<th>Memory Usage</th>
<th>Memory Description Table</th>
<th>Performance</th>
<th>.PDF</th>
<th>Tap Size</th>
<th>Block Size</th>
<th>Coeff’s</th>
<th>create/delete func’s (aka iAlg)</th>
<th>doFilter() (aka iMod)</th>
<th>Inst Obj</th>
<th>block</th>
<th>history</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions to call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*30*

*31*
If given a packaged algo (only object code), what would you need to know to use it?

**Need to Know**

- **Algorithm Parameters**
- **“xDAIS” functions**
- **my DSP functions**

**“Packaged” Algo**

- **Inst Obj**
- **block**
- **history**

**Video Decoder Example**

- **xDM-defined**

**What do you mean, “fixed” params?**

**xDM Interfaces**

- **8 classes:**
  - 4 algo domains: V / I / S / A
  - 2 functionalities: Enc, Dec

- **Enable plug + play ability for multimedia codecs across implementations / vendors / systems**
- **Uniform across domains...video, imaging, audio, speech**
- **Flexibility of extension of 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, 3Ps, and Customers
  - Support Backward Compatibility
Creating alg Interfaces: Component Wizard

Select the Target and Mode of Operation
Information About the Component

Defining Parameters
xDM Video Decoder Data Structures

typedef struct IAUDDEC_Params { // structure used to initialize the algorithm
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 maxSampleRate; // max sampling frequency supported in Hz
    XDAS_Int32 maxBitrate; // max bit-rate supported in bits per secs
    XDAS_Int32 maxNoOfCh; // max number of channels supported
    XDAS_Int32 dataEndianness; // endianness of input data
} IAUDDEC_Params;

typedef struct IVIDDEC_Params { // structure used to initialize the algorithm
    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 as defined in type: XDM_DataFormat
    XDAS_Int32 forceChromaFormat; // set to XDM_DEFAULT to avoid re-sampling
} IVIDDEC_Params;

typedef struct IVIDDEC_DynamicParams { // control API argument
    XDAS_Int32 size; // size of this structure
    XDAS_Int32 decodeHeader; // XDM_DECODE_AU or XDM_PARSE_HEADER
    XDAS_Int32 displayWidth; // default:0. pitch = imagewidth as, else use given display width
    XDAS_Int32 frameSkipMode; // frame skip mode
} IVIDDEC_DynamicParams;

Each of the 8 xDM classes have similar – but differentiated – parameter sets.

Define Memory Blocks – Instance Object
Define Additional Memory Blocks

Define DSP Algorithm Function
Define Algorithm Parameters... Done

For xDM algos the `process()` function is fully predefined, so enter here the protocol of the given xDM algo intended as follows...

**xDM process() API**

```
Int (*process)(IAUDDEC_Handle handle, XDM_BufDesc *inBufs, XDM_BufDesc *outBufs, IAUDDEC_InArgs *inargs, IAUDDEC_OutArgs *outargs);
```

- **handle**: which instance of the algorithm is being called
- **inBufs**: describes input data being passed to algo
- **outBufs**: describes output arrays passed to algo for result storage
- **inargs**: properties of input data
- **outargs**: where info on results are to be stored

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

```c
typedef struct IAUDDEC_InArgs{
    XDAS_Int32 size;
    XDAS_Int32 numBytes;
} IAUDDEC_InArgs;
```

```c
typedef struct IAUDDEC_OutArgs{
    XDAS_Int32 size;
    XDAS_Int32 extendedError;
    XDAS_Int32 bytesConsumed;
} IAUDDEC_OutArgs;
```
Example Audio Decoder Data Structures

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;

xDM Video Decoder Data Structures

typedef struct XDM_BufDesc {
    XDAS_Int32 numBufs;         // number of buffers
    XDAS_Int32 *bufSizes;       // array of sizes of each buffer in 8-bit bytes
    XDAS_Int8 **bufs;           // pointer to vector containing buffer addresses
} XDM_BufDesc;

typedef struct IVIDDEC_InArgs {
    XDAS_Int32 size;            // size of this structure
    XDAS_Int32 numBytes;        // Size of valid input data in bytes in input buffer
    XDAS_Int32 inputID;         // algo tags out frames with this (app provided) ID
} IVIDDEC_InArgs;

typedef struct IVIDDEC_OutArgs {
    XDAS_Int32 size;            // size of this structure
    XDAS_Int32 extendedError;   // extended error code
    XDAS_Int32 bytesConsumed;   // bytes consumed per given call
    XDAS_Int32 decodedFrameType; // frame type
    XDAS_Int32 outputID;        // output ID: tagged w value from *InArgs:InputId
    IVIDEO_BufDesc displayBufs; // buffer pointers for current displayable frames.
} IVIDDEC_OutArgs;
**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 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
  - `XDM_GETSTATUS`: returns status of the last decode call in IAUDDEC_Status structure

- **status**: status of decoder as of the last decode call is written to IAUDDEC_Status structure

**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;**

**xDM Audio Decoder Data Structures**

```c
typedef struct IAUDDEC_Status {
    // used in control API call to relay the status of prior decode
    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;
```

```c
typedef struct XDM_AlgBufInfo {
    // return the size and number of buffers needed for input & output
    XDAS_Int32 minNumInBufs;    // min number of input buffers
    XDAS_Int32 minNumOutBufs;    // min number of output buffers
    XDAS_Int32 minInBufSize[XDM_MAX_IO_BUFFERS];    // min bytes req’d for each input buffer
    XDAS_Int32 minOutBufSize[XDM_MAX_IO_BUFFERS];    // min bytes req’d for ea. output buffer
} XDM_AlgBufInfo;
```
XDM Video Decoder Data Structures

typedef struct XDM_AlgBufInfo {
    XDAS_Int32 minNumInBufs; // min number of input buffers
    XDAS_Int32 minNumOutBufs; // min number of output buffers
    XDAS_Int32 minInBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req’d for each input buffer
    XDAS_Int32 minOutBufSize[XDM_MAX_IO_BUFFERS]; // min bytes req’d for each output buffer
} XDM_AlgBufInfo;

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
    XDAS_Int32 outputChromaFormat; // Chroma output fmt of type IVIDEO_CHROMAFORMAT
    XDM_AlgBufInfo bufInfo; // Input & output buffer info
} IVIDDEC_Status;

Optional: Enter VAB Information

For DaVinci:
Simply click on the “Next” button to
Move on to the following screen…
Ready to “Generate Code”

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

View Code Written by Component Wizard

Reminder:
Now that you have generated your component shell, you need to perform the following steps:
1. Load the project in C Code Composer Studio and add your specific algorithm code.
2. Run the Quality program to verify your component is compliant.
3. Submit your algorithm for testing.

FILE NAME: FIR_TI.h

BEGIN...
LINE 15

END...
Component Wizard Made Instance Object

```c
typedef struct FIR_TI_Obj {
    IALG_Obj        alg;   /* MUST be first field of all FIR objs */
    XDAS_Int16      firLen;
    XDAS_Int16      blockSize;
    XDAS_Int16 *    coeffPtr;
    XDAS_Int16      *workBuffer;
    XDAS_Int16      *historyBuffer;

    /* TODO: add custom fields here */
} FIR_TI_Obj;
```

Component Wizard Made algAlloc()

```c
Int FIR_TI_alloc(const IALG_Params *FIRParams, IALG_Fxns **fxns, IALG_MemRec memTab[]) {
    const IFIR_Params *params = (Void *)FIRParams;
    if (params == NULL) {
        params = &IFIR_PARAMS;  /* set default parameters */
    }

    memTab[0].size = sizeof(FIR_TI_Obj);
    memTab[0].alignment = (4 * 8) / CHAR_BIT;
    memTab[0].space = IALG_SARAM0;
    memTab[0].attrs = IALG_PERSIST;

    memTab[WORKBUFFER].size = (params->firLen+params->blockSize-1) * sizeof(XDAS_Int16);
    memTab[WORKBUFFER].alignment = (2 * 8) / CHAR_BIT;
    memTab[WORKBUFFER].space = IALG_SARAM0;
    memTab[WORKBUFFER].attrs = IALG_SCRATCH;

    memTab[HISTORYBUFFER].size = (params->firLen-1) * sizeof(XDAS_Int16);
    memTab[HISTORYBUFFER].alignment = (2 * 8) / CHAR_BIT;
    memTab[HISTORYBUFFER].space = IALG_EXTERNAL;
    memTab[HISTORYBUFFER].attrs = IALG_PERSIST;

    return (MTAB_NRECS);
}
```
Component Wizard Made algFree()

```c
Int FIR_TI_free(IALG_Handle handle, IALG_MemRec memTab[]) {
    Int n;
    FIR_TI_Obj *FIR = (Void *)handle;

    n = FIR_TI_alloc(NULL, NULL, memTab);

    memTab[WORKBUFFER].base = FIR->workBuffer;
    memTab[WORKBUFFER].size = (FIR->firLen+FIR->blockSize-1) * sizeof(XDAS_Int16);
    memTab[HISTORYBUFFER].base = FIR->historyBuffer;
    memTab[HISTORYBUFFER].size = (FIR->firLen-1) * sizeof(XDAS_Int16);

    return (n);
}
```

Component Wizard Made algInit()

```c
Int FIR_TI_initObj(IALG_Handle handle, const IALG_MemRec memTab[], IALG_Handle p, const IALG_Params *FIRParams) {
    FIR_TI_Obj *FIR = (Void *)handle;
    const IFIR_Params *params = (Void *)FIRParams;

    if(params == NULL){
        params = &IFIR_PARAMS; /* set default parameters */
    }

    FIR->firLen = params->firLen;
    FIR->blockSize = params->blockSize;
    FIR->coeffPtr = params->coeffPtr;
    FIR->workBuffer = memTab[WORKBUFFER].base;
    FIR->historyBuffer = memTab[HISTORYBUFFER].base;

    /* TODO: Implement any additional algInit desired */

    return (IALG_EOK);
}
```
algActivate & algDeactivate Incomplete...

Void FIR_TI_activate(IALG_Handle handle) {
    FIR_TI_Obj *FIR = (Void *)handle;

    // TODO: implement algActivate
    // TODO: Initialize any important scratch memory values to FIR->workBuffer
}

Void FIR_TI_deactivate(IALG_Handle handle) {
    FIR_TI_Obj *FIR = (Void *)handle;

    // TODO: implement algDeactivate
    // TODO: Save any important scratch memory values from FIR->workBuffer
    // to persistent memory.
}

algActivate and 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));
}
Compliant Algorithm “Package”

Compliant algorithms must include:

- Libraries of the code provided
- Header files listing the implemented abstract interfaces
- Documentation defining the algorithm

Algorithm Documentation

<table>
<thead>
<tr>
<th>Module</th>
<th>Vendor</th>
<th>Variant</th>
<th>Architecture</th>
<th>Memory Model</th>
<th>Version</th>
<th>Doc Date</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>TTO</td>
<td>min</td>
<td>62</td>
<td>Little endian</td>
<td>none</td>
<td>04-15-2001</td>
<td>Tms_bd_min_62</td>
</tr>
</tbody>
</table>

ROMable (Rule 1)

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

 Heap Data Memory (Rule 18)

<table>
<thead>
<tr>
<th>memRef</th>
<th>Attribute</th>
<th>Size (bytes)</th>
<th>Align (MAtu)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Persist</td>
<td>20</td>
<td>4</td>
<td>External</td>
</tr>
<tr>
<td>1</td>
<td>Persist</td>
<td>2 <em>(offSet-1) + 2</em>(len-1)</td>
<td>2</td>
<td>DARAM0</td>
</tr>
</tbody>
</table>

Note: This unit for size is 8-256 byte and the unit for align is Minimum Addressable Unit (MAAtu).
DMA and XDAIS

EDMA Hardware Overview

EDMA: Enhanced Direct Memory Access

Goal:
- Simple copy from memory to memory

Examples:
- Import raw data from off-chip to on before processing
- Export results from on-chip to off afterward

Specified by:
- EDMA control registers

EDMA: Elements, Frames, Blocks

Frame size
“BCNT”
In # of
ACNTs

Array
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

Grid

Element size
“ACNT”
In bytes

Block size
“CCNT”
In # of
BCNTs

Source

BCNT

ACNT

Destination

CCNT
### EDMA: Pointer Management?

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

<table>
<thead>
<tr>
<th><code>loc_8</code> (bytes)</th>
<th><code>myDest:</code></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>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
</tbody>
</table>

- DMA always increments across ACNT fields  
- B and C counts must be 1 (or more) for any actions to occur

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ACNT</code></td>
<td><code>BCNT</code></td>
</tr>
<tr>
<td><code>loc_8</code></td>
<td><code>myDest</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>BCNT</code></th>
<th><code>ACNT</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>= 4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>DSTBIDX</code></th>
<th><code>SRCBIDX</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSTCIDX</th>
<th>SRCCIDX</th>
<th>CCNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0</td>
<td></td>
<td>= 1</td>
</tr>
</tbody>
</table>

### EDMA: Pointer Management Alternate

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

<table>
<thead>
<tr>
<th><code>loc_8</code> (bytes)</th>
<th><code>myDest:</code></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>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
</tbody>
</table>

- Here, A count was defined as element size: 1 byte  
- Therefore, B count will now be framesize: 4 bytes  
- B indexing must now be specified as well

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ACNT</code></td>
<td><code>BCNT</code></td>
</tr>
<tr>
<td><code>loc_8</code></td>
<td><code>myDest</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>BCNT</code></th>
<th><code>ACNT</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>= 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>DSTBIDX</code></th>
<th><code>SRCBIDX</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>= 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSTCIDX</th>
<th>SRCCIDX</th>
<th>CCNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0</td>
<td></td>
<td>= 1</td>
</tr>
</tbody>
</table>

**Note:** Less efficient version
**EDMA Example : Indexing**

Goal:
Transfer 4 *vertical* elements from loc_8 to a port

<table>
<thead>
<tr>
<th>loc_8 (bytes)</th>
<th>Codec:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>Codec</td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td></td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
</tr>
</tbody>
</table>

- A count is again defined as element size : 1 byte
- Therefore, B count is still framesize : 4 bytes
- B indexing now will be 6 – skipping to next column

```
4 = Source
   = &loc_8

0 = Destination
   = &myDest

0 = DSTBIDX SRCBIDX
   = 6

0 = DSTCIDX SRCCIDX
   = 0

4 = CCNT
   = 1
```

**EDMA Example : Block Transfer**

Goal:
Transfer a 4x4 subset from loc_8 to a port

<table>
<thead>
<tr>
<th>16-bit Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>7 FRAME 1 12</td>
</tr>
<tr>
<td>13 FRAME 2 18</td>
</tr>
<tr>
<td>19 FRAME 3 24</td>
</tr>
<tr>
<td>25 FRAME 4 30</td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
</tr>
</tbody>
</table>

- A count is here defined as ‘short’ element size : 2 bytes
- B count is again framesize : 4 bytes
- C count now will be 4 – skipping to next column

```
4 = Source
   = &loc_8

0 = Destination
   = &myDest

0 = DSTBIDX SRCBIDX
   = 2

0 = DSTCIDX SRCCIDX
   = 12

4 = CCNT
   = 4
```
**EDMA : Linked Series of Transfers**

**Goal**
Create a list of DMA transfer requests

**Examples**
Continuous Ping-pong buffering

**Specified by**
EDMA LINK control register

### Source
- **BCNT**
- **ACNT**

### Destination
- **DSTBIDX**
- **SRCBIDX**
- **LINK**
- **DSTCIDX**
- **SRCCIDX**
- **CCNT**

---

**EDMA : Linked Series Examples**

**Example 1** Ping-Pong Buffering

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &X [DSTCIDX SRCCIDX CCNT]

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &Y [DSTCIDX SRCCIDX CCNT]

**Example 2** Simple List

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &X [DSTCIDX SRCCIDX CCNT]

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &Y [DSTCIDX SRCCIDX CCNT]

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &Y [DSTCIDX SRCCIDX CCNT]

- &SerPtRcvReg [BCNT ACNT]
- &Ping [DSTBIDX SRCBIDX]
- &Y [DSTCIDX SRCCIDX CCNT]
**EDMA : Multiple Linked Channels**

- There can be up to 8 requestors allowed on DaVinci for asynchronous transfers

<table>
<thead>
<tr>
<th>Requestor 1</th>
<th>Requestor 2</th>
<th>Requestor n</th>
</tr>
</thead>
<tbody>
<tr>
<td>xfer 1</td>
<td>xfer 1</td>
<td>xfer 1</td>
</tr>
<tr>
<td>xfer 2</td>
<td>xfer 2</td>
<td>xfer 2</td>
</tr>
<tr>
<td>xfer 3</td>
<td>xfer 3</td>
<td>xfer n</td>
</tr>
</tbody>
</table>

- How is arbitration managed amongst requests?
  - Priority levels can be assigned to each channel: low or high
  - Within a priority level, transaction requests are queued and served in order received

**How Is The DMA Managed?**

Good: EDMA registers can be programmed directly. Not portable, low level

Better: CSL (Chip Support Library) is a better option – portable, higher level

Best: Algorithm authors can use ACPY API – good abstraction & portability
  - System integrators can tune DMA usage via DMAN management
    (these will be shown in some detail in later chapters)

**Options**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCNT</td>
<td>DSTBINDX</td>
</tr>
<tr>
<td>ACNT</td>
<td>SRCBINDX</td>
</tr>
<tr>
<td></td>
<td>BCNTRLD</td>
</tr>
<tr>
<td></td>
<td>DSTCINDEX</td>
</tr>
<tr>
<td></td>
<td>SRCINDEX</td>
</tr>
<tr>
<td></td>
<td>- rsvid -</td>
</tr>
<tr>
<td></td>
<td>CCNT</td>
</tr>
</tbody>
</table>

**Note:**

- All these abilities employ the QDMA portion of EDMA3
- Ideal for memory to memory operations
- Suited to algorithm support
- Driver support often requires synchronization to a peripheral
  - how can this be implemented?
XDAIS DMA Resource Management

DMA Hardware Resource Management

- Later, optional xDAIS interface
- Allows SI to delegate DMA hardware (channels, PaRam, TCCs) in same fashion as RAM was under original iALG standard to algos
- Algos never ‘take’ DMA HW directly
  - algos tell system its needs (dmaGetChannelCnt(), dmaGetChannels() )
  - SI determines what DMA HW to give/lend to algo (DMAN framework )
  - SI tells algo what DMA HW it may use (dmaInit() )
- SI can give algo DMA HW permanently (persistent) or just when algo is running (scratch)
- Algo author must request scratch resources (ACPY3_activate(), ACPY3_deactivate() )
- No preemption amongst algos sharing a common scratch is permitted
- Scratch management can extend usage of limited DMA HW
  - All algos within a ‘scratch group’ can share same DMA resources
  - Each scratch group / priority level will require its own DMA HW to assure preemption between scratch groups without impairment of DMA activities

Memory / DMA Management Comparison

<table>
<thead>
<tr>
<th>“Framework” DMA Manager (i.e. DMAN3)</th>
<th>“Framework” DMA Manager (i.e. DMAN3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSKT (used by CE)</td>
<td>Used to request/reserve DMA resources (e.g. channels)</td>
</tr>
<tr>
<td>iAlg interface</td>
<td>iDma interface</td>
</tr>
<tr>
<td>memTab</td>
<td>Set of functions &amp; data struct’s for reserving/using DMA channels</td>
</tr>
<tr>
<td>dmaTab</td>
<td>dmaTab</td>
</tr>
<tr>
<td>Specifies attributes of required DMA resources</td>
<td></td>
</tr>
<tr>
<td>algActivate</td>
<td>ACPY3_activate</td>
</tr>
<tr>
<td>“process” function</td>
<td>ACPY3_start</td>
</tr>
<tr>
<td>algDeactivate for shared (i.e. scratch) memory</td>
<td>Starts async copy by QDMA</td>
</tr>
<tr>
<td></td>
<td>ACPY3_deactivate for shared/scratch DMA channels</td>
</tr>
</tbody>
</table>
DMAN3 / IDMA3 / ACPY3 Interaction

Application

Framework

DMA Manager Module (DMAN3)

ACPY3 Library

IDMA3 Interface

Algorithm

DMA Hardware

needs handle(s)
ACPY3 API

ACPY3 Method Chronology

**ACPY3_init**  - ready to do ACPY3 work

- **ACPY3_activate**  - obtain rights to channel(s)
- **ACPY3_configure**  - describe DMA actions to perform
- **ACPY3_start**  - begin DMA work
- **ACPY3_wait()**  - optional CPU jobs in // w DMA
- **ACPY3_fastConfigure16**
- **ACPY3_start**
- **ACPY3_wait()**

**ACPY3_deactivate**  - release rights to channel(s)

**ACPY3_exit**

*Note: ACPY3_wait is a spin loop (not a block) – enforces resource holding – don’t want to allow some other SG member to run and override currently in use resources…*

ACPY3 Interface

<table>
<thead>
<tr>
<th>ACPY3 Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPY3_init</td>
<td>Initialize the ACPY3 module</td>
</tr>
<tr>
<td>ACPY3_activate</td>
<td>Activate individual DMA channel before using</td>
</tr>
<tr>
<td>ACPY3_configure</td>
<td>Configure a logical channel</td>
</tr>
<tr>
<td>ACPY3_fastConfigure16b</td>
<td>Modify a single (16-bit) PaRameter of the logical DMA channel</td>
</tr>
<tr>
<td>ACPY3_fastConfigure32b</td>
<td>Modify a single (32-bit) PaRameter of the logical DMA channel</td>
</tr>
<tr>
<td>ACPY3_start</td>
<td>Submit dma transfer request using current channel settings</td>
</tr>
<tr>
<td>ACPY3_wait</td>
<td>Wait for all transfers to complete on a specific logical channel</td>
</tr>
<tr>
<td>ACPY3_waitLinked</td>
<td>Wait for an individual transfer to complete on logical channel</td>
</tr>
<tr>
<td>ACPY3_complete</td>
<td>Check if the data 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>
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 DMA Transfer Abstraction

<table>
<thead>
<tr>
<th>ACPY3 PaRam</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transferType</td>
<td>Transfer type: 1D1D, 1D2D, 2D1D or 2D2D</td>
</tr>
<tr>
<td>srcAddr</td>
<td>Source Address of the DMA transfer</td>
</tr>
<tr>
<td>dstAddr</td>
<td>Destination Address of the DMA transfer</td>
</tr>
<tr>
<td>elementSize</td>
<td>Number of consecutive bytes in each 1D transfer vector (ACNT)</td>
</tr>
<tr>
<td>numElements</td>
<td>Number of 1D vectors in 2D transfers (BCNT)</td>
</tr>
<tr>
<td>numFrames</td>
<td>Number of 2D frames in 3D transfers (CCNT)</td>
</tr>
<tr>
<td>srcElementIndex</td>
<td>Offset (in bytes) between starts of each 1D vector (SBIDX)</td>
</tr>
<tr>
<td>dstElementIndex</td>
<td>Offset (in bytes) between starts of each 1D vector (DBIDX)</td>
</tr>
<tr>
<td>srcFrameIndex</td>
<td>Offset in number of bytes from beginning of the first 1D vector of source frame to the beginning of the first element in the next frame. (SCIDX): signed value between -32768 and 32767.</td>
</tr>
<tr>
<td>dstFrameIndex</td>
<td>Offset in number of bytes from beginning 1D vector of first element in destination frame to the beginning of the first element in next frame (DCIDX): signed value between -32768 and 32767.</td>
</tr>
<tr>
<td>waitId</td>
<td>For a linked transfer entry: -1: no individual wait on this transfer 0 &lt;= waitId &lt; numWaits : this transfer can be waited on or polled for completion. Ignored for single-transfers, which are always synchronized with waitid=0.</td>
</tr>
</tbody>
</table>
### ACPY3_configure Example

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

![Diagram](image)

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

### ACPY3_configure Example Code

```c
MOD_VEN_Obj *hMyAlgo = (Void *)handle;
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(hMyAlgo->hMyDma, &PaRam);
ACPY3_start(hMyAlgo->hMyDma);
```
**ACPY3_fastConfigure32b, 16b**

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

void ACPY3_fastConfigure16b (IDMA3_Handle handle, ACPY3_PaRamField16b fieldId, unsigned short value, short transferNo);

- This is a fast configuration function for modifying existing channel settings.
- Exactly one 16 (32)-bit channel transfer property, corresponding to the specified ACPY3_PaRam field, can be modified.
- Remaining settings of the channels configuration are unchanged.

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

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;
```

**ACPY3_activate, ACPY3_deactivate**

extern void ACPY3_activate(IDMA3_Handle hdl);

- ACPY3_activate must be called once before submitting DMA transfers on any IDMA3 channel.
- The executing task remains the active user of all shared scratch resources, and thus should not be pre-empted by any other task that uses DMA resources created using the same Group Id.

extern void ACPY3_deactivate(IDMA3_Handle hdl);

- ACPY3_deactivate is called when a task is ready to relinquish ownership of the shared scratch resources.
**ACPY3_start**

```c
extern void ACPY3_start(IDMA3_Handle hdl);
```

- This function is called to submit a single or linked DMA transfer. The properties of each individual transfer is the most recent configuration setting for that transfer.

**ACPY3_wait, _waitLinked**

```c
extern void ACPY3_wait(IDMA3_Handle hdl);
```

- Wait for all data transfers on a logical channel to complete. ACPY3_wait() uses waitId '0' to **wait for the completion of all transfers**.
- Therefore, waitId '0' should not be used to configure any intermediate transfers.

```c
extern void ACPY3_waitLinked(IDMA3_Handle hdl, unsigned short waitId);
```

- **ACPY3_wait** function performs a polling wait operation, waiting for the completion of an individual DMA transfer issued by the most recent ACPY3_start operation on the given channel handle.
- The transfer that gets waited on is the individual transfer that was configured with the associated waitId.
**ACPY3_complete, _completeLinked**

```c
extern int ACPY3_complete(IDMA3_Handle hdl);
```

- **ACPY3_complete** is the non-blocking counterpart of the **ACPY3_wait** function.
- It returns true or false depending on whether all transfers have already completed or not, respectively.

```c
extern int ACPY3_completeLinked(IDMA3_Handle hdl, unsigned short waitId);
```

- **ACPY3_complete** is the non-blocking counterpart of **ACPY3_waitLinked**.
- It returns true or false depending on whether the individual transfer associated with the waitId have already completed or not, respectively.

**ACPY3_setFinal**

```c
extern int ACPY3_setFinal(IDMA3_Handle hdl, MdInt transferNo);
```

- Indicate that a given transfer will be the last in a sequence of linked transfers.
- This API can be used if a channel was created to transfer numTransfers linked transfers, but at some point, it may be that fewer transfers than numTransfers should be started.
While many DMA channels are provided on DaVinci, there is a limit. Eventually, a system may have enough components competing for resources that all channels are taken.

At that point, what happens? Usually, there’s no clean failure – instead, there’s misbehaviour of the components as they overwrite each other’s resources, making debug very difficult.

What are these resources?

- **Channels**: a channel is a register set in the DMA that specifies a transaction to be implemented. There are 8 channels of QDMA on DaVinci DM6446.

- **TCCs**: a TCC is a Transfer Count Complete status/interrupt signal. There are 64 TCCs on DaVinci. Some may be assigned to the ARM and the rest for the DSP. Usually, all DSP TCCs are given to DMAN to manage.

- **PaRam**: PaRam is a ‘fast copy’ structure, usually in L1D internal RAM. The DSP CPU can write to these quickly, saving many CPU cycles compared to writing to the channel registers over the peripheral bus (20 cycles vs 1). Once setup is written to PaRam, an internal DMA transaction copies these to the channel registers, offloading the 20 cycles per write effort from the DSP CPU. This is why DMAN requests an internal buffer and if not available, DMA setup time goes up significantly.

## iDMA API

<table>
<thead>
<tr>
<th>Logical DMA chan</th>
<th>iDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPY3_start()</td>
<td>algo needs one PaRameter set for each DMA transfer it requires</td>
</tr>
<tr>
<td></td>
<td>algo needs one tcc (transfer complete code) for each synchronization point</td>
</tr>
<tr>
<td></td>
<td>algo needs a logical DMA channel</td>
</tr>
<tr>
<td>ACPY3_wait()</td>
<td>uses one tcc</td>
</tr>
</tbody>
</table>

- While many DMA channels are provided on DaVinci, there is a limit. Eventually, a system may have enough components competing for resources that all channels are taken.
- At that point, what happens? Usually, there’s no clean failure – instead, there’s misbehaviour of the components as they overwrite each other’s resources, making debug very difficult.
- What are these resources?

- **Channels**: a channel is a register set in the DMA that specifies a transaction to be implemented. There are 8 channels of QDMA on DaVinci DM6446.
- **TCCs**: a TCC is a Transfer Count Complete status/interrupt signal. There are 64 TCCs on DaVinci. Some may be assigned to the ARM and the rest for the DSP. Usually, all DSP TCCs are given to DMAN to manage.
- **PaRam**: PaRam is a ‘fast copy’ structure, usually in L1D internal RAM. The DSP CPU can write to these quickly, saving many CPU cycles compared to writing to the channel registers over the peripheral bus (20 cycles vs 1). Once setup is written to PaRam, an internal DMA transaction copies these to the channel registers, offloading the 20 cycles per write effort from the DSP CPU. This is why DMAN requests an internal buffer and if not available, DMA setup time goes up significantly.

## IDMA interface functions

```c
typedef struct IDMA3_Fxns {
    Void *implementationId;
    Void (*dmaChangeChannels)(IALG_Handle, IDMA3_ChannelRec *);
    Int (*dmaGetChannelCnt)(Void);
    Int (*dmaGetChannels)(IALG_Handle, IDMA3_ChannelRec *);
    Int (*dmaInit)(IALG_Handle, IDMA3_ChannelRec *);
} IDMA3_Fxns;

typedef struct IDMA3_ChannelRec {
    IDMA3_Handle handle; // handle to logical DMA chan
    Int numTransfers; // = 1 (single) or N linked xfers
    Int numWaits; // # TCCs needed
    Int envSize; //
    IDMA3_Priority priority; // Urgent, High, Medium, Low
    IDMA3_InterfaceType protocol; // ACPY3_PROTOCOL, other
} IDMA3_ChannelRec;
```

**Logical Flow:**

- `dmaGetChannelCnt()`: returns the number of DMA channels that the algo needs (i.e. size of dma table array)
- `dmaGetChannels()`: fills in the dma table passed to it with the dma channel requirements, and if it has been granted, channels, the associated handles currently allocated in its object
- `dmaInit()`: apps call this to grant dma handle(s) to the algorithm at initialization. Often the same as `dmaChangeChannels`
- `dmaChangeChannels()`: sets new values from dma table into algorithm object (note IDMA3_ChannelRec * is an array, i.e. table, not ptr)
### dmaGetChannelCount, dmaGetChannels

```c
Int MOD_VEN_dmaGetChannelCnt(Void) {
    return(3);
}
```

**Return the number of channels required by the algorithm.**

```c
Int MOD_VEN_dmaGetChannels(IALSE_Handle handle, IDMA3_ChannelRec dmaTab[ ])
{
    MOD_VEN_Obj *fcpy = (Void *)handle;
    int i;
    dmaTab[0].handle = fcpy->hChan1;
    dmaTab[1].handle = fcpy->hChan2;
    dmaTab[2].handle = fcpy->hChan3;
    for (i=0 ; i < 3 ; i++)
    {
        dmaTab[i].numTransfers = 1;
        dmaTab[i].numWaits = 1;
        dmaTab[i].priority = IDMA3_PRIORITY_LOW;
        dmaTab[i].protocol = &ACPY3_PROTOCOL;
        dmaTab[i].persistent = FALSE;
    }
    return (3);
}
```

**Fill the referenced channel descriptors table with the channel characteristics for each logical channel required by the algorithm.**

### dmaInit, dmaChangeChannels

**Save the handles for the logical channels granted by the framework in the algorithm instance object's persistent memory.**

```c
Int MOD_VEN_dmaInit(IALSE_Handle handle, IDMA3_ChannelRec dmaTab[ ])
{
    MOD_VEN_Obj *fcpy = (Void *)handle; // de-ref inst obj. handle
    // read each handle granted into instance object fields
    fcpy->hChan1 = dmaTab[CHANNEL0].handle;
    fcpy->hChan2 = dmaTab[CHANNEL1].handle;
    fcpy->hChan3 = dmaTab[CHANNEL2].handle;
    return (IALG_EOK);
}
```

**Update the algorithm instance object's persistent memory via the channel descriptor table**

```c
Void MOD_VEN_dmaChangeChannels(IALSE_Handle handle, IDMA3_ChannelRec dmaTab[ ])
{
    MOD_VEN_Obj *fcpy = (Void *)handle;
    fcpy->hChan1 = dmaTab[CHANNEL0].handle;
    fcpy->hChan2 = dmaTab[CHANNEL1].handle;
    fcpy->hChan3 = dmaTab[CHANNEL2].handle;
}
```
**DMA and XDAIS**

## DMAN3 Framework

### DMAN3 Concepts

- **Physical DMA Channel**
  - EDMA PaRam 1
  - EDMA PaRam 2
  - EDMA PaRam 3
  - EDMA PaRam 4
- **Logical DMA 1**
  - PaRam 1
  - PaRam 2
  - PaRam 3
  - PaRam 4
- **Logical DMA 2**
  - PaRam 1
  - PaRam 2

- **Scratch channels** allow reuse of PaRam's, DMA channels and TCC's between algorithms
- **Analogous to scratch memory sharing in xDAIS**
- **Algorithms which share DMA resources cannot preempt each other**

### DMAN3 Framework

- **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
  - Physical DMA chans

#### Usage Phase (run-time)

- **Alg1:**
  - 2 PaRam,
  - 2 TCC
  - 1 DMA ch

- **Alg2:**
  - 2 PaRam,
  - 1 TCC
  - 1 DMA ch

### DMAN3 Description

<table>
<thead>
<tr>
<th>DMAN3 method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMAN3_createChannels</td>
<td>allocate DMA resources for new logical channel</td>
</tr>
<tr>
<td>DMAN3_grantDmaChannels</td>
<td>grant logical channel resources to an algo</td>
</tr>
<tr>
<td>DMAN3_freeChannels</td>
<td>free DMA resources used to create logical channel</td>
</tr>
<tr>
<td>DMAN3_releaseDmaChannels</td>
<td>take back logical channel resource from an algo</td>
</tr>
</tbody>
</table>

- **These API are managed by the Codec Engine in DaVinci systems**
- **They are presented here “FYI” for Codec Engine users**
- **Non Codec Engine systems can use these API directly to manage xDAIS algos**
Conclusions

Introduction

In this chapter a number of design support products and services offered by TI to assist you in the development of your DSP system will be described.

Objectives

As initially stated in module 1, you should now be able to:

• Define key software design challenges in developing real-time systems
• Demonstrate essential skills in the use of Code Composer Studio (CCS) in authoring a real-time system
• Identify and apply the optimal DSP/BIOS constructs to implement a given real-time system
• Analyze and optimize a software solution to meet real-time requirements

Module Topics

Conclusions ...............................................................................................................................................14-1
DSP/BIOS Summary ..............................................................................................................................14-2
Development Tools........................................................................................................................... 14-3
For More Information..........................................................................................................................14-4
DSP/BIOS Summary

DSP/BIOS Summary

- Object based programming
- Real-time instrumentation
- Preemptive interrupt scheduling
  - Allows for reentrant code to be used by multiple threads (reentrant code cannot modify: global or static variables, or itself without protection)
  - Overhead
    - Memory for stack and objects
    - Context switching
  - Inter-thread communication and synchronization
  - Basic interrupt handling capabilities
- Real-time data communications with the host
- Support for meeting timing requirements
  - Maintains an optional real-time clock
  - Provides a method to trigger periodic functions
  - Threads can invoke API for measuring performance and optimizations
- Minimize run-time overhead
  - Generated optimized runtime code
  - Predictable context switching times
  - Minimizes interrupt latency

CCS: Orthogonal Software Development

- Code Composer
  - get the code to work...
    - Single Algorithm
    - Single Channel
      - Single GUI for Develop & Debug
      - Graphical Data Analysis
      - Optimizing C Compiler
      - Expandable via Plug-ins
- DSP/BIOS
  - meet real-time goals...
    - concurrent multi algorithm
      - Prioritized Preemptive Thread Scheduling
      - Real-Time Analysis – Debug w/o halt
      - Hardware Abstraction – Easier system s/u

Code Composer Studio - separate tools to independently solve different problems!
Development Tools

DSK Packages...

Documentation
- DSK Technical Ref.
- eXpressDSP for Dummies

Software
- Code Composer Studio
- SD Diagnostic Utility
- Example Programs

Hardware
- 1 GHz C6416 DSP
  or 225 MHz C6713 DSP
- TI 24-bit A/D Converter (AIC23)
- External Memory
  - 8 or 16M Bytes SDRAM
  - Flash ROM - C6416 has 512K Bytes
    - C6713 has 256K Bytes
- LED’s and DIP’s
- Daughter card expansion
- 1 or 2 additional expansions
- Power Supply & USB Cable
For More Information

**TI Website : www.ti.com**

![TI Website](http://www.ti.com)

**TI Documentation via ti.com**

*from Ti.com, select: Technical Documents / App Notes (Users Guides, etc) / DSP*
TI Documentation - via “dspvillage”


DSP/BIOS™ Real-Time OS

● View DSP/BIOS™ Kernel Overview

DSP/BIOS™ is a scalable real-time kernel, designed specifically for the TMS320C6000™ and TMS320C6600™ DSP platforms. DSP/BIOS has been proven in thousands of customer designs and is included as part of the Code Composer Studio™ Development Tools. DSP/BIOS requires no runtime license fees and is backed by Texas Instruments worldwide training and support organization.

DSP/BIOS enables you to develop and deploy sophisticated applications more quickly than with traditional DSP software methodologies and eliminates the need to develop and maintain custom operating systems or control loops. Because multi-threading enables near real-time applications to be cleanly partitioned, an application using DSP/BIOS is easier to maintain and new functions can be added without disrupting real-time response. DSP/BIOS provides standardized APIs across C6000 and C6600 DSP platforms to support rapid application migration.

Related Links
- Development Tools
- Code Composer Studio
- Development Tools: DSP/BIOS
- CSP License Agreement
- Product Bulletin: DSP/BIOS
- Tool List
- Customer Success

TI Documentation - via CCS

◆ From CCS: select “Help” and “Users Manuals”

TMS320C6000 Code Composer Studio Manuals

Software Documentation
DSP Foundation Software
Hardware Documentation
TMS320 DSP Algorithm Standard Documentation
Application Reports

Click here for the latest user guides

Software Documentation

Use this document: SPRU150 Code Composer Studio Getting Started Guide

If you need information about:

Provided basic procedures in prod you begin programming Code Com

Getting started quickly.

Getting started quickly.

Getting started quickly.

How to use DSP/BIOS to embed

Assembly language

DSP/BIOS - Conclusions 14 - 5
### Example BIOS & C6000 Documentation

<table>
<thead>
<tr>
<th>Category</th>
<th>ID Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSP/BIOS</strong></td>
<td>SPRA782</td>
<td>How to Get Started with the DSP/BIOS Kernel</td>
</tr>
<tr>
<td></td>
<td>SPRA780</td>
<td>DSP/BIOS Kernel Technical Overview</td>
</tr>
<tr>
<td></td>
<td>SPRA640</td>
<td>Programming and Debugging Tips for DSP/BIOS</td>
</tr>
<tr>
<td></td>
<td>SPRA900</td>
<td>DSP/BIOS Timing Benchmarks for CCS 2.2</td>
</tr>
<tr>
<td></td>
<td>SPRA772</td>
<td>DSP/BIOS Sizing Guidelines on TMS320C6000/C5000 for CCS 2.2</td>
</tr>
<tr>
<td></td>
<td>SPRA829</td>
<td>DSP/BIOS Timers and Benchmarking Tips</td>
</tr>
<tr>
<td></td>
<td>SPRA660</td>
<td>Building DSP/BIOS Programs in UNIX</td>
</tr>
<tr>
<td></td>
<td>SPRA653</td>
<td>Understanding Basic DSP/BIOS Features</td>
</tr>
<tr>
<td></td>
<td>SPRA599</td>
<td>DSP/BIOS and TMS320C54X Extended Addressing</td>
</tr>
<tr>
<td></td>
<td>SPRA783</td>
<td>DSP/BIOS by Degrees: Using DSP/BIOS in an existing application</td>
</tr>
<tr>
<td><strong>C6000</strong></td>
<td>SPRU328</td>
<td>Code Composer Studio User's Guide</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>SPRU423</td>
<td>TMS320 DSP/BIOS User's Guide</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>SPRU403</td>
<td>TMS320C6000 DSP/BIOS API Reference Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU401</td>
<td>TMS320C6000 Chip Support Library API Reference Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU187</td>
<td>TMS320C6000 Optimizing C Compiler User's Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU186</td>
<td>TMS320C6000 Assembly Language Tools User's Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU402</td>
<td>TMS320C62x DSP Library Programmer's Reference</td>
</tr>
<tr>
<td><strong>Devices</strong></td>
<td>SPRU189</td>
<td>TMS320C6000 CPU and Instruction Set Reference Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU190</td>
<td>TMS320C6000 Peripherals Reference</td>
</tr>
<tr>
<td></td>
<td>SPRU197</td>
<td>TMS320C6000 Technical Brief</td>
</tr>
<tr>
<td></td>
<td>SPRU198</td>
<td>TMS320C62/C67X Programmer's Guide</td>
</tr>
</tbody>
</table>

### Example C5xxx Documentation

<table>
<thead>
<tr>
<th>Category</th>
<th>ID Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5xxx</strong></td>
<td>SPRU328</td>
<td>Code Composer Studio User's Guide</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>SPRU423</td>
<td>TMS320 DSP/BIOS User's Guide</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>SPRU404</td>
<td>TMS320C5000 DSP/BIOS API Reference Guide</td>
</tr>
<tr>
<td><strong>55xx</strong></td>
<td>SPRU433</td>
<td>TMS320C55x Chip Support Library API User's Guide</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>SPRU422</td>
<td>TMS320C55x DSP Library Programmer's Reference</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>SPRU280</td>
<td>TMS320C55x Assembly Language Tools User's Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU281</td>
<td>TMS320C54x Optimizing C/C++ Compiler User's Guide</td>
</tr>
<tr>
<td><strong>54xx</strong></td>
<td>SPRU420</td>
<td>TMS320C54x Chip Support Library API User's Guide</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>SPRA480</td>
<td>Optimized DSP Library for C Programmers on the ‘C54x</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>SPRU102</td>
<td>TMS320C54x Assembly Language Tools User's Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU103</td>
<td>TMS320C54x Optimizing C/C++ Compiler User's Guide</td>
</tr>
<tr>
<td><strong>55x</strong></td>
<td>SPRU371</td>
<td>TMS320C55x DSP CPU Reference Guide</td>
</tr>
<tr>
<td><strong>Devices</strong></td>
<td>SPRU374</td>
<td>TMS320C55x DSP Mnemonic Instruction Set Reference Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU375</td>
<td>TMS320C55x DSP Algebraic Instruction Set Reference Guide</td>
</tr>
<tr>
<td></td>
<td>SPRU317</td>
<td>TMS320C55x DSP Peripherals Reference Guide</td>
</tr>
<tr>
<td><strong>54x</strong></td>
<td>SPRU371</td>
<td>TMS320C55x DSP Reference: CPU and Peripherals</td>
</tr>
<tr>
<td><strong>Devices</strong></td>
<td>SPRU172</td>
<td>TMS320C54x DSP Reference: Mnemonic Instruction Set</td>
</tr>
<tr>
<td></td>
<td>SPRU179</td>
<td>TMS320C54x DSP Reference: Algebraic Instruction Set</td>
</tr>
<tr>
<td></td>
<td>SPRU173</td>
<td>TMS320C54x DSP Reference: Applications Guide</td>
</tr>
</tbody>
</table>
## One Day Workshops Offered by TI

From TI.com, select: Training / By Type / 1-day workshops

Educational programs designed to offer attendees training on DSP products and development tools. Developed as a mini-workshop, this training usually lasts one day and includes a "hands-on" section utilizing a development tool. These workshops are facilitated by TI Field Sales representatives and are ideal for developers who are getting started with DSP technology.

### 1-DAY WORKSHOPS

<table>
<thead>
<tr>
<th>Workshop Description</th>
<th>Applications/DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video and Audio Applications Design Hands-on Workshop based on TMS320DM642</td>
<td>Applications/DSP</td>
</tr>
<tr>
<td>Evaluating DSP for Embedded Applications Workshop</td>
<td>DSP</td>
</tr>
<tr>
<td>Digital Motor Control One-Day Workshop</td>
<td>DSP</td>
</tr>
<tr>
<td>TMS320C5510 DSK One-Day Workshop</td>
<td>DSP</td>
</tr>
<tr>
<td>TMS320FP2812 ezdsp One-Day Workshop</td>
<td>DSP</td>
</tr>
<tr>
<td>DSP/BIOS (TM) OS One-Day Workshop</td>
<td>Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C6416/6713 DSK One-Day Workshop</td>
<td>DSP</td>
</tr>
</tbody>
</table>

### ARCHIVED 1-DAY WORKSHOPS

<table>
<thead>
<tr>
<th>Workshop Description</th>
<th>Applications/DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using FPGAs and DSPs Together for Real-Time Processing</td>
<td>DSP</td>
</tr>
<tr>
<td>Implementing Signal Processing Applications With Programmable DSPs</td>
<td>DSP</td>
</tr>
<tr>
<td>eXpressDSP (TM) One-Day Workshop</td>
<td>Applications</td>
</tr>
<tr>
<td>Implementation of Video Streaming One-Day Application Workshop</td>
<td>Applications</td>
</tr>
</tbody>
</table>

[http://focus.ti.com/docs/training/catalog/events/eventsbytype.jhtml?templateId=5517&navigationId=8460](http://focus.ti.com/docs/training/catalog/events/eventsbytype.jhtml?templateId=5517&navigationId=8460)

---

## Full Workshops Offered by TI

From TI.com, select: Training / By Type / Multi-day workshops

Advanced educational programs designed for engineers who need to sharpen their design and development skills. Workshops usually last two to five days and include significant "hands-on" sections emphasizing the demonstration and application of techniques and skills. TI Workshops are given by TI's Technical Training Staff and are highly beneficial in helping developers implement their DSP designs quickly.

### MULTI-DAY WORKSHOPS

<table>
<thead>
<tr>
<th>Workshop Description</th>
<th>DSP / OMAP / Tools &amp; Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMAP (tm) Software Workshop</td>
<td>DSP / OMAP / Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C33x (tm) DSP Integration Workshop</td>
<td>DSP / Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C54x (tm) DSP Integration Workshop</td>
<td>DSP / Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C6000 (tm) DSP Integration Workshop</td>
<td>DSP / Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C28x (tm) DSP Workshop</td>
<td>DSP</td>
</tr>
<tr>
<td>DSP/BIOS (tm) OS Design Workshop</td>
<td>Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C5500 (tm) DSP Optimization Workshop</td>
<td>DSP / Tools &amp; Software</td>
</tr>
<tr>
<td>TMS320C24x (tm) DSP Workshop</td>
<td>DSP / Tools &amp; Software</td>
</tr>
</tbody>
</table>

[http://focus.ti.com/docs/training/catalog/events/eventsbytype.jhtml?templateId=5517&navigationId=8461](http://focus.ti.com/docs/training/catalog/events/eventsbytype.jhtml?templateId=5517&navigationId=8461)

---

Sign up by clicking on desired workshop / register now / select region / select class
For More Information . . .

Internet
Website: http://www.ti.com
    http://www.dspvillage.com
FAQ:
    http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm
    • Device information
    • Application notes
    • Technical documentation
    • my.ti.com
    • News and events
    • Training
Enroll in Technical Training: http://www.ti.com/sc/training

USA - Product Information Center (PIC)
Phone: 800-477-8924 or 972-644-5580
Email: support@ti.com
    • Information and support for all TI Semiconductor products/tools
    • Submit suggestions and errata for tools, silicon and documents

Visit the DSP Village for the latest DSP/BIOS info.

Reference Literature on DSP

◆ “A Simple Approach to Digital Signal Processing”
  by Craig Marven and Gillian Ewers; ISBN 0-4711-5243-9

◆ ”DSP Primer (Primer Series)”


◆ ”DSP First: A Multimedia Approach (Matlab Curriculum Series)”
  James H. McClellan; ISBN 0-1324-3171-8
Thank You For Attending!

TTO
Technical Training Organization

Texas Instruments
Appendix

Acronyms

API  Application Process Interface – defined protocol to interact with software components
BIOS  or “DSP/BIOS”- RTOS for TI DSPs
BSL  Board Support library – extension of CSL for a given target board
CCS  Code Composer Studio – IDE for developing DSP software
CSL  Chip Support Library – interface to peripherals via predefined structures and API
CXD  Create / Execute / Delete - three phases of the life cycle of a node on OMAP under Bridge
DARAM  Dual Access RAM – RAM internal to the DSP cell that allows 2 transactions per cycle
DIO  Device IO – interface between an IOM and SIO
DIP  Dual In-line Position switch – simple on/off switch, several of which are found on the DSK
DMA  Direct Memory Access – sub-processor that manages data copy from one place in memory to another
DSK  DSP Starter Kit – low-cost evaluation tool for developing DSP solutions
DSP  Digital Signal Processor - processor with enhanced numerical abilities
EDMA  Enhanced Direct Memory Access – DMA controller with advanced options, more channels than before
EMIF  External Memory Interface – TI DSP sub-component that manages flow of data on/off chip
EVM  Evaluation Module – hardware test and debug platform
FIR  Finite Impulse Response – non-recursive filter
GEL  General Extension Language – macro/batch file for use in CCS
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
ICE  In Circuit Emulator – hardware debug tool
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
LED  Light Emitting Diode – low power bulb
MAC  Multiply Accumulate – core activity in many DSP algorithms
MAU  Minimum Allocation Unit – smallest element that can be malloc’d on a given processor
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>McBSP</td>
<td>Multi Channel Buffered Serial Port – serial port that interfaces to numerous data formats</td>
</tr>
<tr>
<td>MIPS</td>
<td>Millions of Instructions Per Second – basic measure of DSP performance</td>
</tr>
<tr>
<td>MP3</td>
<td>MPEG 4 level 3 – video standard audio encoding methodology</td>
</tr>
<tr>
<td>MPEG</td>
<td>Motion Picture Experts Group – video compression standard</td>
</tr>
<tr>
<td>MPU</td>
<td>Micro Processor Unit – processor core</td>
</tr>
<tr>
<td>NMADUs</td>
<td>Number of Minimum Addressable Data Units</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer – maker of a given hardware solution</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System – software that provides sophisticated services to software authors</td>
</tr>
<tr>
<td>POST</td>
<td>Power On Self Test – DSK diagnostic routine that runs on reset of DSP</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory – memory that can be read from or written to</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer – processor with small, fast instruction set</td>
</tr>
<tr>
<td>RTA</td>
<td>Real-Time Analysis – ability to observe activity on a processor without halting the device</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System – software kernel that is tuned to deterministic temporal behavior</td>
</tr>
<tr>
<td>SARAM</td>
<td>Single Access RAM – RAM internal to the DSP cell that allows a transaction each cycle</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous DRAM – clock driven (fast) Dynamic Random Access Memory</td>
</tr>
<tr>
<td>SISO</td>
<td>Single Input Single Output – most basic DSP system, see also IPO</td>
</tr>
<tr>
<td>SW</td>
<td>Software – code run on hardware</td>
</tr>
<tr>
<td>TI</td>
<td>Texas Instruments – semiconductor manufacturer specializing in DSP and analog</td>
</tr>
<tr>
<td>TLA</td>
<td>Three Letter Acronym – examples all over these pages...</td>
</tr>
<tr>
<td>TTO</td>
<td>Texas Instruments Training Organization – group within TI chartered to provide DSP training</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter – serial port with clock embedded in data</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus – modern interface between PCs, handhelds, and numerous peripherals</td>
</tr>
<tr>
<td>XDAIS</td>
<td>eXpress DSP Algorithm Interface Standard – rules that define how to author DSP components</td>
</tr>
<tr>
<td>XDS</td>
<td>eXtended Development System – TI hardware platform DSP development system</td>
</tr>
</tbody>
</table>
**Pre-class Questionnaire**

**Name (optional)__________________________**

---

**Questionnaire**

*In the table below, for each topic please rank your relative prior experience and your current interest (need to learn) on a 0 (low) to 9 (high) point scale.*

<table>
<thead>
<tr>
<th>Experience</th>
<th>Interest</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Real-Time System Considerations</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Hardware – General</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSP Architecture: <em>which ones?</em>_________________</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPP Processor: <em>which ones?</em>____________________</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware (Board) Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drivers</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Software – General</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASM Coding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Coding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++ / OOP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code Composer Studio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSP Algorithm Standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSP/BIOS (check which you’ve used before:)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ☐ SEM ☐ TSK ☐ SWI ☐ PIP ☐ SIO ☐ MEM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other RTOS: <em>which ones?</em>_______________________</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Real-Time System Overview</strong></td>
</tr>
</tbody>
</table>

---

Please note any other information you’d like the instructor to consider below:

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________