Application Report
SPRA783A - September 2001

DSP/BIOS by Degrees: Using DSP/BIOS (CCStudio 2.0) Features in an Existing Application

Thom Maughan, Kathryn Rafac, Mohammad Khatami *Software Development Systems*

ABSTRACT

DSP/BIOS™ provides a complete set of instrumented kernel services optimized for fast execution speed and small size for use with Texas Instruments digital signal processors. DSP/BIOS is instrumented to provide real time analysis of software executing on TI DSPs. DSP/BIOS is included in Code Composer Studio™ along with the editor, compiler, project manager, and debugger. This introduces a revolutionary approach to writing and analyzing real time software applications. DSP/BIOS provides a priority-based scheduler along with a set of real time analysis (RTA) tools, and real-time data exchange (RTDX) between the host computer and the target DSP.

Real-time application design is a balance of hardware and software that achieves the best performance within a particular cost. Embedded systems often require real time analysis for debugging complex applications. DSP/BIOS provides powerful tools which make the challenges of real time system design easier so they can meet critical deadlines. This application note provides an example of the steps for integrating DSP/BIOS features into an existing application.

Contents

1 Introduction

The integration of DSP/BIOS into an application can be performed in multiple steps or degrees. The following degrees demonstrates this implementation:

- **Zero Degree: 'The Starting Point'.** This degree introduces a non-BIOS program, 'audio', that uses the Multichannel Buffered Serial Port (McBSP) and the Enhanced Direct Memory Access (EDMA) to continuously process digitized audio through the codec on the DSK6211 (it also works with DSK6711) target board.
- **First Degree: 'Real Time Printf Debugging'.** The first degree covers the steps necessary to use the LOG_printf() function in the audio application. LOG_printf() provides printf() debugging with minimal impact on code size and execution time.
- **Second Degree: 'New Ways to Look at Data'.** The second degree demonstrates the steps required to incorporate DSP/BIOS statistics (STS) objects and RTDX into the application. The examples show how STS objects are used to profile execution time and to gather statistics on variables and other program conditions at run time. The second degree also demonstrates hardware interrupt logging.
- **Third Degree: 'Going All the Way'.** The third degree demonstrates how to use the DSP/BIOS scheduler in order to get full access to the real time analysis features and the benefits of using a standard coding infrastructure.

2 Zero Degree: The Starting Point

This application note uses an existing, non-BIOS application to demonstrate the steps required to incorporate DSP/BIOS using Code Composer Studio 2.0. The application, audio, is built to run on TI's DSK6211 (or DSK6711) board and uses the Codec, Multichannel Buffered Serial Port (McBSP) and enhanced direct memory access (EDMA) to continuously process digitized audio data. This application is built to execute out of SBSRAM and to use SDRAM for data storage. .

The structure of the application is shown in Figure 1.

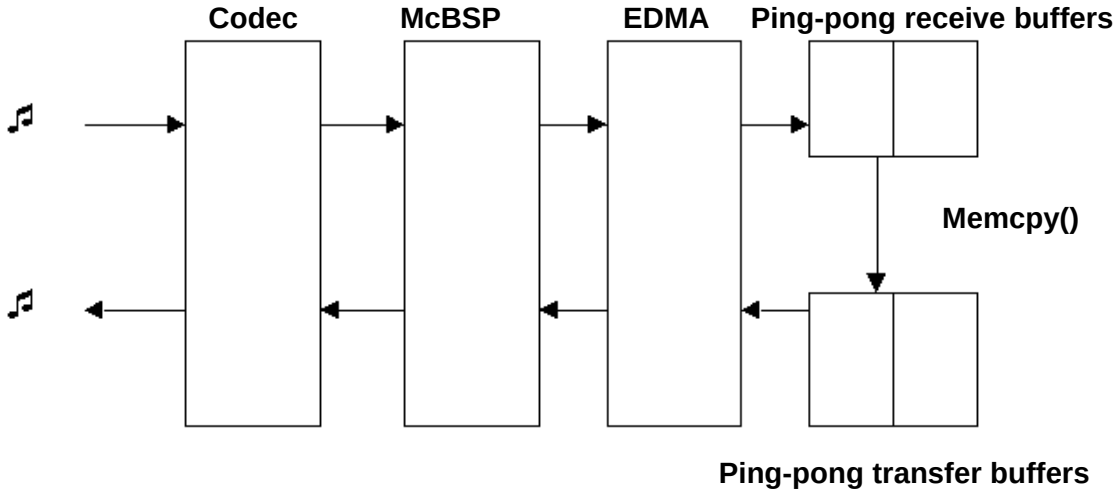


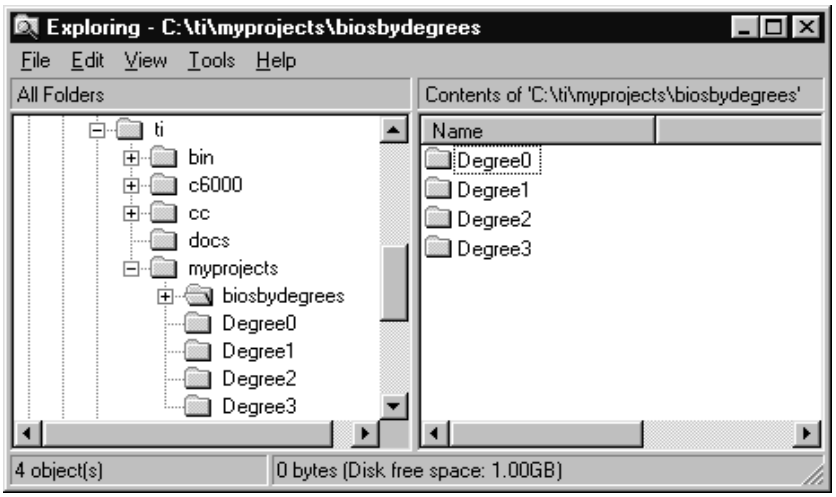
Figure 1. Audio Application

Complete source code is provided with this document. The source code is available from the Appendix A and also from the accompanying zip files. While the Appendix A contains the source code for Degree0 only, the zip files contain the four projects: Degree0, Degree1, Degree2 and Degree3. Each one of these projects contains the state of the audio application after all the instructions in the corresponding chapter have been completed.

In order to get the most out of the examples, it is recommended that you work through the instructions in the text. To do so, either cut and copy the code from the Appendix A to create Degree0 project, or unzip the accompanying zip files, copy only the Degree0 project folder intoC:\ti\myprojects\biosbydegrees, and only use the other project folders for reference.

The remainder of this chapter is instructions on how to create the Degree0 project folder if you choose to cut and copy the code from the Appendix A. If you are working with the unzipped Degree0 folder, you can move on to The First Degree.

Degree Zero shows the necessary steps to collect the pre-existing source files from Appendix A into a project in order to create your starting point. In every degree, you create a new folder and copy all files from the previous degree into the new folder. The end result is a directory structure with a folder for each degree you have completed. The project in each folder represents the project state at the end of the degree. When you have finished all four degrees, the directory structure will look like this.



Step A: Create the necessary source files.

Cut and copy the source codes from Appendix A:

1. Create the directory C:\ti\myprojects\biosbydegrees.
2. In the biosbydegrees folder, make a new folder called Degree0.
3. Browse to Appendix A at the end of this document. Appendix A contains the source code for audio.c, device.c, vectors.asm, linker.cmd, coeffs.c and audio.h. For each of these files, copy the code into a text editor such as WordPad. (To select text in Adobe Acrobat Reader, click on the button on the toolbar or press the letter 'V' on the keyboard to switch into text select mode. Then select the text that you wish to copy.)
4. Save the files into the Degree0 directory. When you finish, the Degree0 directory should contain the six files: audio.c, device.c, coeffs.c, vectors.asm, linker.cmd and audio.h.

NOTE: Pasting from this document into a text editor converts quotes and minus signs into block characters that Code Composer Studio does not recognize. To solve this problem, copy a block character (for example, from the printf() in audio.c) into the Edit –> Replace function of your text editor. (Depending on the text editor you are using, these characters may or may not appear as blocks, but they still need to be replaced in order for Code Composer Studio to recognize them when you open the source files in Code Compuser Studio.) Replace all block characters with the double quotation mark in audio.c, device.c and vectors.asm. Also, in the edmaEnable() function in device.c, search for the assignment statements that include minus signs that become block characters in Code Composer Studio. Replace them with minus signs.

NOTE: Pasting from this document also results in a loss of all indentation that causes problems in vectors.asm because assembly files require that only labels appear in the first column. In vectors.asm, scroll through the text and insert tabs in all lines of text that do not begin with labels. (Reference Appendix A to determine which lines need to be indented.)

Step B: Create the audio.pjt project.

Now that you have the source files, assemble them into a Code Composer Studio project.

1. Double-click on the Code Composer Studio icon. Select Project –> New. In the Location box, browse to Degree0 folder. In the Project Name box, type audio. In the Project Type, choose the .out file and click finish.

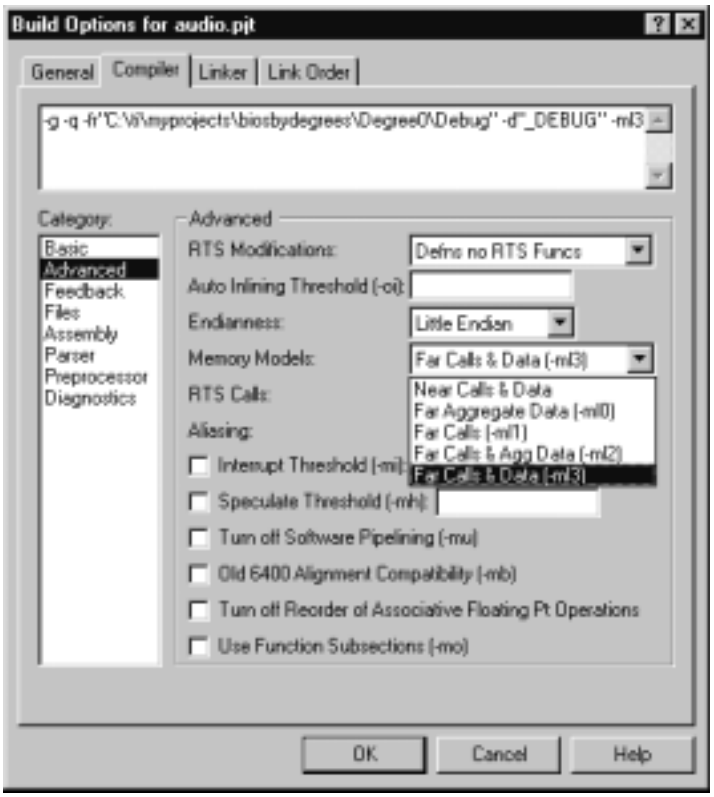
2. Add the source files to your project. Choose Project –> Add Files to Project. Choose audio.c and click Open. Do the same with device.c, coeffs.c and vectors.asm.

3. Choose Project –> Add Files to Project. Select Linker Command Files (*.cmd) in the Files of type box. Now choose linker.cmd and click Open.

4. Choose Project –> Add Files to Project. Select Library Files (*.lib) in the Files of type box. Browse to the C:\ti\c6000\bios\lib directory. Choose csl6211.lib and click Open. Repeat the same process for the following library files: C:\ti\c6000\dsplib\lib\dsp62x.lib and C:\ti\c6000\cgtools\lib\rts6201.lib. It is not necessary to add audio.h to the project because it is automatically included during the project build.

5. Choose Project –> Build Options. Select Compiler tab and add the following paths:

   –i”C:\ti\c6000\dsplib\include”, –i”c:\ti\c6000\bios\include” and –d”CHIP_6211”. Click OK.



6. The program executes out of SDRAM, so it is a good idea to reset the external memory interface (EMIF) when you launch Code Composer Studio and between executions of the program. Choose Debug –> Reset CPU.

7. Choose Project –> Rebuild All to build the audio project. Load the program by choosing File –> Load Program and selecting audio.out.

8. In order to hear the audio, you must have a radio or CD player and powered speakers connected to the DSK6211 board with mini-jack cables. Choose Debug –> Go Main. This sets a temporary breakpoint at the beginning of the main function. The program runs until it hits main(). It is not necessary to use Debug –> Go Main before executing your programs but it is convenient to do so, for it automatically loads the main()source code into the text editor. Choose Debug –> Run to run the program and hear the results.

9. Save the project before moving to the next degree.

3 The First Degree: Real-Time Printf Debugging

Replacing the printf() with LOG_printf() results in significant code size reduction. Table 1 shows the comparison between the size and speed of LOG_printf() versus printf(). LOG_printf() enables a print-debugging technique in places where printf() would interfere with program execution. LOG_printf() achieves such a low impact on speed and size by relying on the host to perform the string formatting. Each LOG_printf() transfers four words to the host where significant COFF file processing results in the formatted data being displayed in the associated DSP/BIOS Message Log. Additional information on using LOG_printf() is available in Code Composer Studio's Help menu.

Table 1. Size / Speed Comparison of LOG_printf and printf

	Size (bytes)	Speed (instruction cycles)
printf()	37,848	631
LOG_printf()	132*	36

* This includes a 32 byte buffer.

In general, addition of LOG_printf() to an existing application requires the following procedure:

- Step A: Create a DSP/BIOS configuration file.
- Step B: Insert a LOG object into the configuration file.
- Step C: Migrate the interrupt vector table into the HWI module of the configuration file.
- Step D: Migrate memory segments and sections from the linker command file into the MEM module of the configuration file.
- Step E: Save the configuration file and update the project file.
- Step F: Create a compound linker command file.
- Step G: Add BIOS_start() to initialize and IDL_run() to pump the real time data link.
- Step H: Replace printf() with LOG_printf().

The use of LOG_printf() requires creating a DSP/BIOS configuration. The configuration is used to create the LOG object and the necessary link between the host and target for real time data analysis. When the configuration is saved, the DSP/BIOS configuration file, *.cdb, generates five files: *cfg.s62, *cfg.h62, audiocfg.h, audiocfg_c.c and *cfg.cmd. The *cfg.s62 and its associated header file, *cfg.h62, contain the necessary objects and the hardware interrupt vector table. The linker command file, *cfg.cmd, is responsible for including the appropriate DSP/BIOS libraries as well as including the run-time support library (RTS6201.lib).

The following steps show the creation of the DSP/BIOS configuration and the resulting files. They demonstrate creating the LOG object and mapping the memory regions and the interrupt vector table into the Configuration Tool. Lastly, they show the code which uses the LOG object with LOG_printf() to replace the original printf() function.

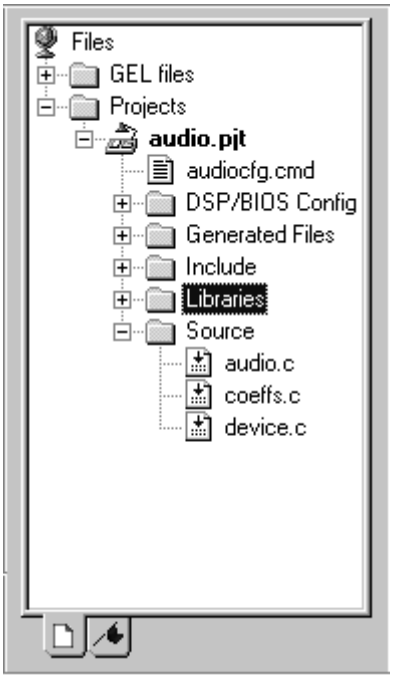
Step A: Create a DSP/BIOS configuration file.

The DSP/BIOS Configuration Tool is first used to configure a LOG object. The unused DSP/BIOS modules do not impact code size; however, the RTDX data pump is automatically included because it transports the LOG_printf() data from the target to the host.

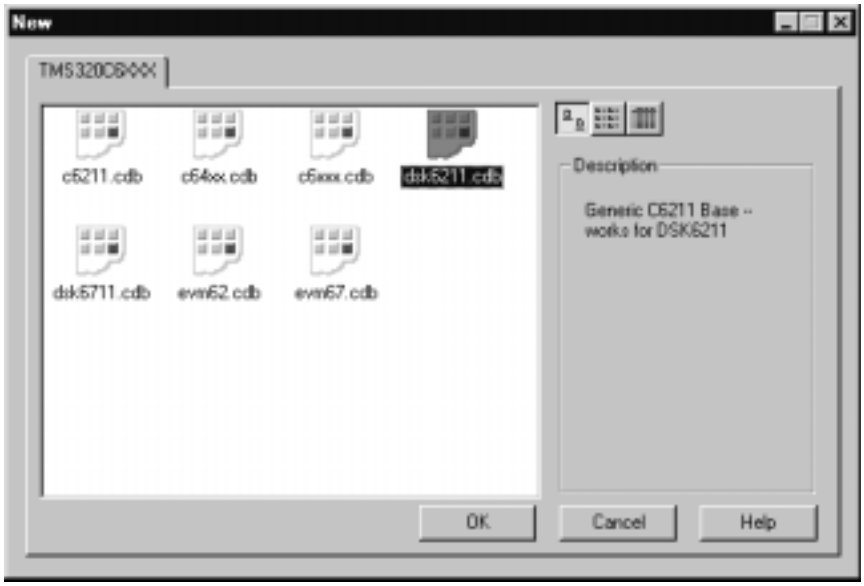
1. In c:\ti\myprojects\biosbydegrees, create a new folder called Degree1. Copy all the files from Degree0 into Degree1.
2. Double-click on the Code Composer Studio icon on the desktop. Choose Project –> Open and browse to the Degree1 folder. Select the audio.pjt file and click Open. If you are working with Degree0 from the zip files, you will receive a dialog box with the message that the library,

rts6201.lib cannot be found. This is because the project has been moved. Choose the Browse button on the dialog box, and browse to the file in C:\ti\c6000\cgtools\lib.

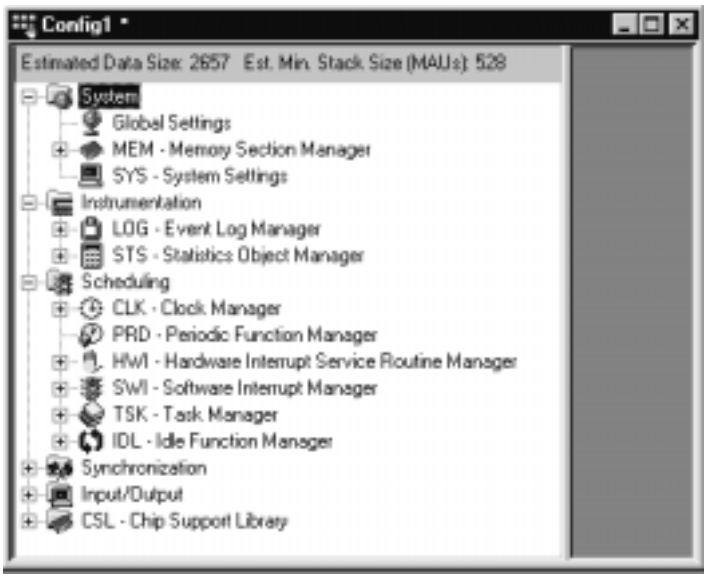
3. Click on the + sign next to the Project folder in the Project View and then click on the + sign next to audio.pjt. This expands the project. Click on the + sign next to the source folder to see all the source files associated with the application.



4. Choose File –> New –> DSP/BIOS Config. Select the configuration template entitled dsk6211.cdb and click OK.



5. This presents the DSP/BIOS Configuration Tool. You use the Configuration Tool to create your configuration file and to incorporate DSP/BIOS modules into the application.

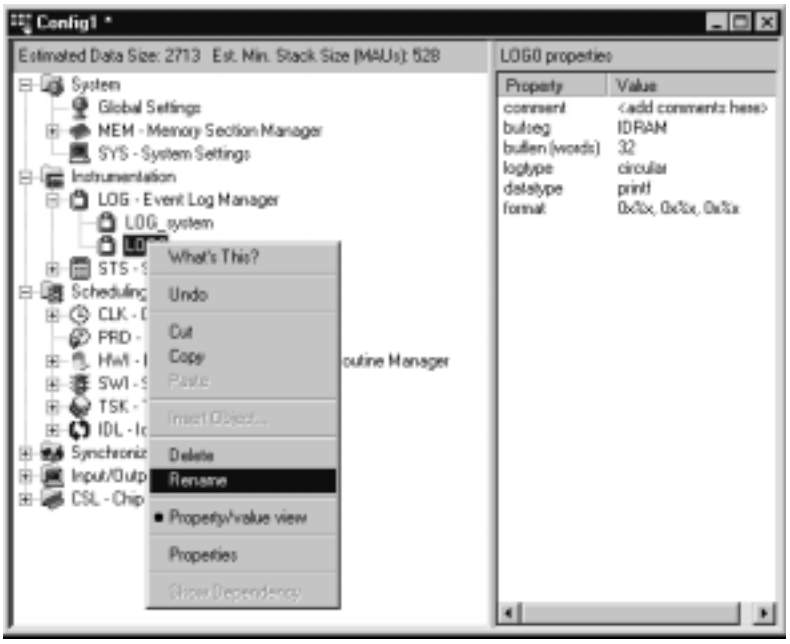


Step B: Insert a LOG object into the configuration file.

The goal is to define a LOG object for managing the LOG_printf() messages.

1. Click on the + sign next to the Instrumentation. Right-click on Event Log Manager. Choose Insert LOG from the pull-down menu. This results in the creation of a new LOG object called LOG0.

2. Right-click on LOG0 and choose Rename from the pull-down menu. Rename the object trace.



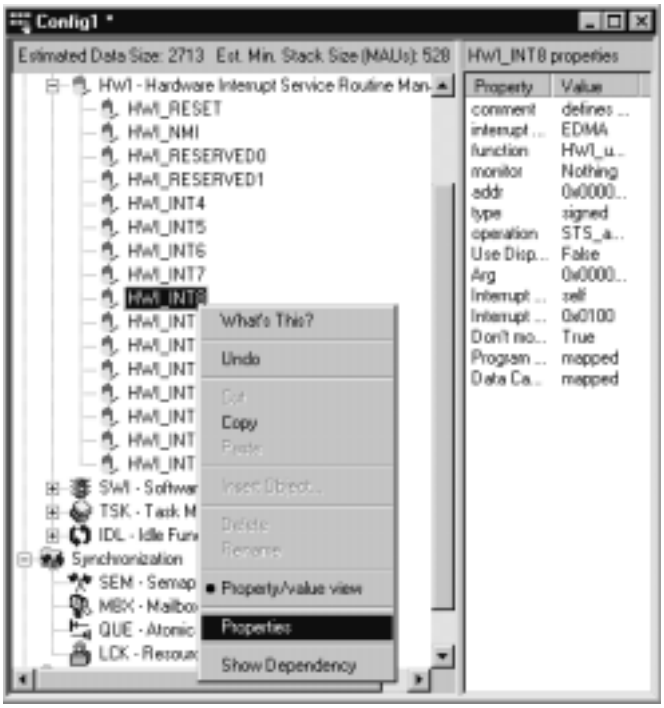
3. Right-click on trace and choose Properties from the pull-down menu. Change the buflen property to 512 words and click OK. This buffer holds (512 words) / (4 words per LOG_printf) = 128 LOG_printf messages in a circular fashion.

Step C: Migrate the interrupt vector table to the HWI module of the configuration file.

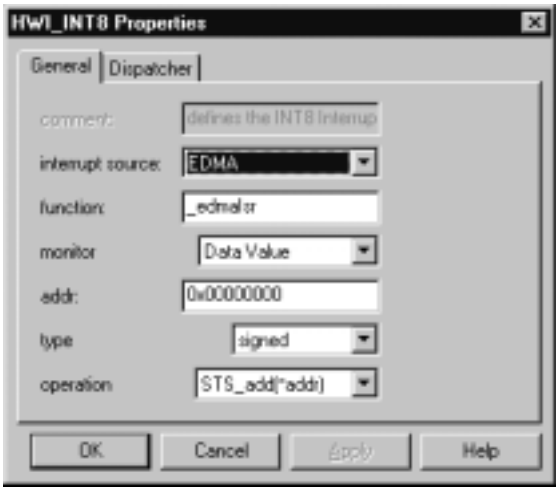
Each handled interrupt must be migrated into the HWI section of the Configuration Tool.

1. Vectors.asm is the assembly language file that defines the interrupt vector table for the audio application. Double-click on the vectors.asm file. Examine the code and notice that only interrupts 8 and reset are handled by interrupt service routines (ISR). The EDMA receive ISR, edmaIsr, uses interrupt 8. Close the file.

2. In the Configuration Tool, click on the + sign next to the Scheduling. Click on the + sign next to the Hardware Interrupt Service Routine Manager to expand the list of available interrupts. To designate an ISR for hardware interrupt 8, right-click on HWI_INT8 and choose Properties from the pop-up menu.



3. In the properties window, type the name of the appropriate interrupt service routine, "_edmaIsr", in the function text box. Notice that the C function names must be preceded with an underscore. Make sure that the interrupt source is mapped to the correct device. In this case the default assignment, EDMA, is correct. Click OK.

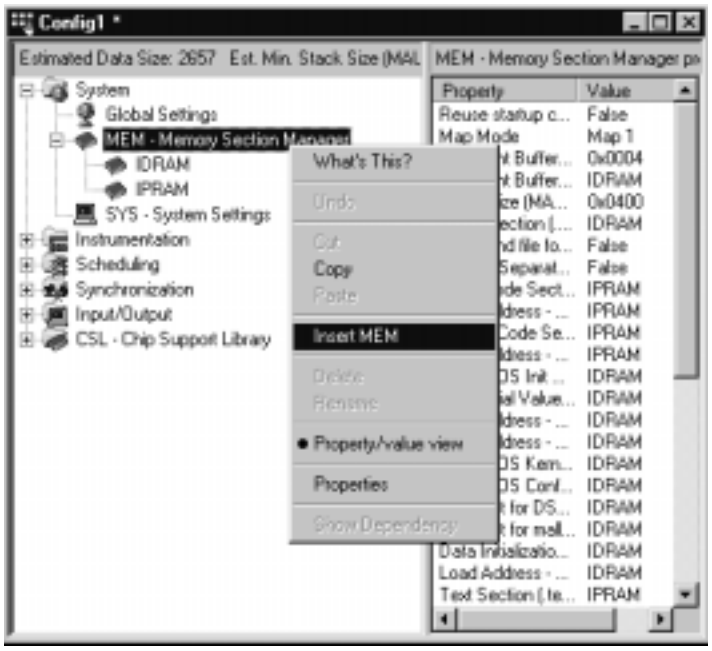


4. Click on the – sign next to Hardware Interrupt Service Routine Manager to collapse the list of interrupts.

Step D: Migrate memory sections from Linker command file to configuration MEM module.

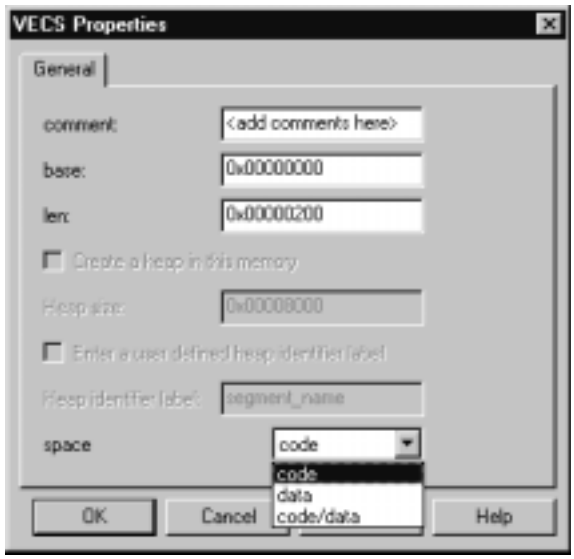
Use the Configuration Tool to define common memory regions and migrate definitions from the application's original linker command file.

1. Use a text editor to open the application's linker command file, linker.cmd. Notice the memory regions defined within the MEMORY directive, including the name, base address, and length of each region. These specifications need to be migrated into the DSP/BIOS configuration.
2. Return to the Configuration Tool and click on the + sign next to the System. Click on the + sign next to the Memory Section Manager. The Memory Section Manager has a predefined memory segment, SDRAM. This segment cannot be renamed or removed.
3. Right-click on SDRAM. Choose Properties from the pop-up menu. Confirm that SDRAM has a base address of 0x80000000 and a length of 0x01000000 and a Heap size of 0x00008000. Click OK.
4. Right-click on the Memory Section Manager in the Configuration Tool and select Insert MEM from the pop-up menu.
5. Right-click on the Global Settings and select Properties from the pop-up menu. Select the 621x/671x tab. In the L2 Mode – CCFG(L2MODE) label, select SRAM from the drop down menu. Press OK.



6. Right-click on the new memory segment and choose Rename from the pop-up menu. Rename the memory object VECS. Right-click on VECS and choose Properties from the pop-up menu.

7. In the properties window uncheck the check box labeled Create a heap in this memory. Set the base equal to 0x00000000 and the length to 0x00000200. Select code to designate the type of memory space you are defining. Click OK. Delete the VECS definition from the linker command file.



8. Continue to transfer the following memory region into the Configuration Tool as you did in steps 4, 6, and 7: IRAM. Select code/data for the type of memory space. ***Note***: Do not check the box labeled 'create a heap in this memory'. Set its base address to 0x00000200 and its length to 0x0000FE00. Delete this element from the linker command file once it is in the Configuration Tool. This results in an empty MEMORY directive. Delete the MEMORY directive from the linker command file.

9. Refer to the linker command file, linker.cmd, and notice the assignments specified in the SECTIONS directive. To assign sections to particular regions in memory, return to the Configuration Tool and right-click on the Memory Section Manager. Choose Properties from the pop-up menu. The various tab sections enables you to assign sections to areas of memory. (These assignments designate run as well as load addresses and do not permit you to determine a separate load and run address.)

10. Assign the Text Section (.text), the BIOS Code Section (.bios) and the Startup Code Section (.sysinit) to IRAM.

11. Assign the Data Section (.data, .switch, .cio, .far, .const, .bss, .stack, .cinit) to IRAM.

12. Assign the Data Initialization Section(.pinit) and TRC initial Value to SDRAM. Assign the interrupt Service Table Memory(.hwi_vec) to VECS.

13. If it is possible to remove all section assignments from within the SECTION directive of Linker.cmd file, then remove the Linker.cmd file from the project and skip Step F on next page.

Step E: Save the configuration file and update the project file.

Saving the configuration generates five files: audiocfg.s62, audiocfg.h62, audiocfg.h, audiocfg_c.c and audiocfg.cmd. The audiocfg.s62 file contains the assembly language code necessary to configure the interrupt vector table and the DSP/BIOS LOG object, as well as the RTDX data pump which transfers data between the target and the host. The audiocfg.cmd file defines all the necessary memory segments and sections required by the rts6201.lib and DSP/BIOS libraries. Be aware that the configuration file has to reside in the same directory as the final executable file.

1. Choose File–>Save. Name the configuration file audio.cdb and save it into the Degree1 folder. Click Save. Close the Configuration Tool by choosing File–>Close.

2. Now that you have created and saved the configuration, you need to add the resulting files to the project and remove the files that they replace. Choose Project –> Add Files to Project. Choose Configuration File (*.cdb) in the Files of type box. Select audio.cdb file and click Open. As a result, audiocfg.s62 is also added to the project.

3. If the vectors.asm file is open in the text editor, close it. Then, in the Project View, right-click on the vectors.asm file. Choose Remove from project from the pop-up menu. This file is no longer necessary because you defined the interrupt vectors in the configuration file.

4. Click on the + sign next to the Libraries folder. Right-click on rts6201.lib. Choose Remove from project from the pop-up menu. Also remove the library file, csl6211.lib. Recall that the new audiocfg.cmd file already includes this library.

5. If you removed the Linker.cmd file from the project on Step D, choose Project –> Add Files to Project. Select audiocfg.cmd and click open.

6. If in some applications, the Linker.cmd file has not been removed from the project, follow the Step F.

Step F: Create a compound linker command file.

If the entire contents of the application's linker command file had been completely migrated to the DSP/BIOS configuration, then the *cfg.cmd file that is created as a result of the DSP/BIOS configuration can simply be added to the existing project to replace the original linker command file. When the situation arises that special cases remain in the application linker command file, a compound linker command file is necessary. You cannot add a second linker command file to a project, but a compound command file enables you to specify additional command files as linker input. This step demonstrates the simplicity of creating a compound linker command file. The end result is a linker command file that calls the DSP/BIOS command file.

1. Add "–laudiocfg.cmd" (without the double quotes) as the first line of code in the original linker command file. The –l flag (lowercase L) is the linker library switch. By making this the first line of your file, you ensure that the DSP/BIOS linker command definitions precede those in the application's linker command file.

2. Save the file and exit the text editor.

Step G: Add BIOS_start() to initialize and IDL_run() to pump the real time data link.

BIOS_start() enables hardware interrupts and initializes the data pipes used by the real time analysis tools. BIOS_start() also provides initialization code that is required to utilize DSP/BIOS modules, such as the SWI (Software Interrupt) Module, and to execute BIOS API calls such as IDL_run(). IDL_run() executes the idle functions configured under the IDL Manager. These functions can be seen in the Configuration Tool by clicking the + sign next to the IDL Function Manager. In the audio project, only the default functions are present, but you may use the Configuration Tool to add idle functions as necessary. The default idle functions are responsible for moving the RTDX data between the target and host and for calculating processor utilization for the DSP/BIOS CPU Load tool. Calling IDL_run() is necessary to take advantage of real time analysis features such as LOG_printf(). The LOG_printf() messages buffered on the DSP are transferred by the LNK_dataPump() function to the host PC for processing and display in a Message Log tool. IDL_run() must execute multiple times in order to pump data over the RTDX real time link between the host and target.

Because the executive loop of the audio project is still contained within the main() function, it is necessary to call BIOS_start() and IDL_run() explicitly. However, in Degree Three, you rewrite the audio project so that the program returns from main() and relies upon the scheduler to manage execution of the functions previously contained in the executive loop in main(). Once the functionality of the program is removed from main() and managed by the scheduler through software and hardware interrupts, calling BIOS_start and IDL_run() is no longer necessary because they are called automatically after execution returns from main(). The file, boot.c, which is found in the C:\ti\c6000\bios\src\misc folder, illustrates this. If you look at the end of the boot.c file, you see that after the user's main() function executes, BIOS_start() runs, and then IDL_loop() runs (which is similar to the IDL_run() function.) IDL_loop() causes execution to loop through the idle functions until interrupted either by a hardware, software interrupt, or task. (See the Third Degree for details on the scheduler.)

NOTE: While IDL_run() makes it possible to use DSP/BIOS tools such as the Message Log and Statistics View by pumping the data from the target to host, the DSP/BIOS CPU Load tool should not be used in programs that do not return from main(). The CPU Load Tool will not produce accurate results because its calibration depends on full use of the scheduler.

1. In the Configuration Tool, right-click on the Task Manager and select Properties. Uncheck the box labeled Enable Task Manager. Click OK. Save the configuration and close the file. In the next step, you add a call to BIOS_start() within your source code. Disabling the Task Manager is necessary to ensure that the call to BIOS_start returns. See the DSP/BIOS User's Guide for more information on the DSP/BIOS startup sequence.

2. Double-click on audio.c in the Project View in order to open the file. Insert a call to BIOS_start() as the first line in the main() function after the variable declarations.

3. Add a call to IDL_run() within the executive loop as shown in the example below.

```
while(1) {
     if (inputReadyFlag && outputReadyFlag) {
         inputReadyFlag = 0; /* Reset flags          */
         outputReadyFlag = 0;
         processBuffer();    /* Process Data         */
     }
     IDL_run();
}
```

4. Comment out the IRQ_globalRestore(1) from interruptsEnable function in device.c. This call is no longer necessary because BIOS_start() enables the global interrupt enable (GIE) and non-maskable interrupt (NMI). When incorporating DSP/BIOS into any program where interrupts are used, it is important not to explicitly enable interrupts or the NMI, but to allow DSP/BIOS to handle interrupt initialization. The only initialization that is still necessary is to enable the specific interrupts that are used by your application. For example, in audio, mapping the DMA event to HWI_INT8 is done in interruptsEnable function.

Step H: Replace printf() with LOG_printf().

Use LOG_printf() to replace the printf() function in the audio example.

1. In order to use the log object created in the Configuration Tool, it is necessary to include the std.h and log.h header files. Each DSP/BIOS module requires using a corresponding header file. In addition, the DSP/BIOS header files should start with std.h. Add these file inclusions to audio.c. Because printf() is no longer being used, you can also remove the stdio.h file inclusion.

```
#include <std.h>
#include <log.h>
#include ″audio.h″
```

2. Declare an external reference to the log object, trace. Add the declaration above the main function in audio.c.

```
/* LOG Object created by the Configuration Tool */
extern far LOG_Obj trace;
```

3. In audio.c insert call to LOG_printf() as shown in the example below. Remember to delete the original printf() function call. Save audio.c when you are done.

```
void main()
{
   int date = 25;

   BIOS_start();
   CSL_init();
   initApplication();      /* Initialise DSP System   */
   interruptsEnable();


   LOG_printf(&trace, "Why do computer scientists celebrate Halloween and
     Christmas on the same day?\
     \nBecause oct %o  equals dec %d.\n", date, date);
```

Step I: Run the program.

1. Choose Project –> Rebuild All.
2. Choose Debug –> Reset CPU.
3. Choose File –> Load Program and select audio.out. Click Open.
4. Choose DSP/BIOS –> Message Log. Select trace from the drop-down list. Click OK. The Message Log now enables you to view the messages generated by the LOG_printf() function calls.
5. Choose Debug –> Go Main, and then Debug –> Run.
6. Save the project before moving to the next degree.

4 The Second Degree: New Ways to Look at Data

4.1 Analyzing Variables, Execution Time, and Custom Data Displays

The second step in incorporating DSP/BIOS into the application is to utilize the statistics objects. The Statistics Object Manager in the Configuration Tool enables you to create custom statistics objects that have the flexibility to be designed to suit a variety of purposes. For example, statistics objects are suitable for monitoring the occurrence of an event, tracking the maximum or minimum value of a variable through time, or determining execution time. Statistics monitoring, like LOG_printf(), interferes very little with the execution of the program. All statistics gathering is done in real time with minimal space requirements on the target. The results are displayed in the Statistics View real time analysis tool.

Degree Two demonstrates the steps required to add statistics objects to the application and to view the results. The examples illustrate some of the benefits of using the STS objects provided by DSP/BIOS to profile execution times and to gather statistics on variables. The last example shows the use of HWI monitoring to determine interrupts per second.

Step A: Use statistics to watch a variable through time.

This step utilizes a statistics object to monitor the range of results from a sine wave generator.

1. In C:\ti\myprojects\biosbydegrees, create a new folder called Degree2. Copy all the files from Degree1 into Degree2.
2. Double-click on the Code Composer Studio icon on the desktop. Choose Project –> Open and browse to the Degree2 folder. Select the audio.pjt file and choose Open.

3. In the Project View, click on the + sign next to the Project folder and then the + sign next to audio.pjt. Click on the + sign next to the DSP/BIOS Config folder and Source folder.

4. Double-click on audio.c to open the source file. Add the sine wave generator function, generateSine() as shown below, toward the end of the file. The sine() function returns the sine of the radian variable into the sinValue variable. Add a call to generateSine() in the main() function right after the LOG_printf() function.

```
/*
 * ======== generateSine ========
 * Sine Wave Generator. Applicable in Degree2
 */

void generateSine(void)
{
    double sinValue;
    double radian = 0;
    int index = 0;

    for(index = 0; index < 100; index ++){
          sinValue = sin(radian);
          radian = radian + PI/90;
    }
}
```

5. Add the following header files to audio.c. Remember that this addition must be included after std.h.

```
#include <string.h>
#include <math.h>
```

6. Double-click on audio.cdb to open the Configuration Tool.

7. Click on the + sign next to the Instrumentation. Right-click on the Statistics Object Manager and choose Insert STS from the pop-up menu.

8. Right-click on the new STS object, STS0, and choose Rename from the pop-up menu. Rename the object stsSinMax. Right-click again on the Statistics Object Manager and choose Insert STS. Rename the new object stsSinMin.

9. Choose File –> Save to save the changes to the configuration. Close the tool.

10. Add the necessary DSP/BIOS header file to audio.c. Remember that this addition must be included after std.h.

```
#include <sts.h>
```

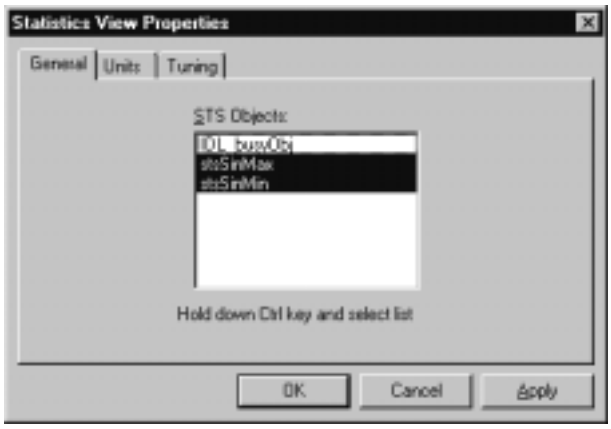
11. Declare the statistics objects in audio.c by adding the following lines above the main() function. Now that the entire program is stored in internal memory, the far keyword is no longer necessary; however, it is still used so that if the code is moved later, it can be done so without encountering problems with displacement.

```
extern far STS_Obj stsSinMax;
extern far STS_Obj stsSinMin;
```

12. In the generateSine() function, add calls to STS_add() below the call to the sin() function as shown in the example below. The result is multiplied by 1000 before passing it into STS_add() because STS_add() expects a long integer and the sinValue is a double. STS_add() is useful for tracking a variable through time because it updates the statistics object's Count, Max, and Total fields using the data value provided.

```
for(index = 0; index < 100; index++){
    sinValue = sin(radian);
    STS_add(&stsSinMax, (1000 * sinValue));
    STS_add(&stsSinMin, -(1000 * sinValue));
    radian = radian + PI/90;
}
```

13. Choose File –> Save.
14. Choose Debug –> Reset CPU.
15. Choose Project –> Rebuild All and load the executable audio.out.
16. Choose DSP/BIOS –> Statistics View. Right-click on the Statistics View area and choose Property Page from the pop-up menu. Select the stsSinMax and stsSinMin objects. Click OK.



17. Choose Debug –> Go Main and then Debug > Run.
18. Notice the values in the Statistics View. (Recall that the sine values have been multiplied by 1000) Count reports the number of times STS_add() was called for the statistics objects. For stsSinMax, Max is the maximum value that sinValue reached and Average is the average sine value. For stsSinMin, Max is the negative of the minimum value that sinValue reached. To clear the Statistics View, right-click on the Statistics View area and select Clear from the pop-up menu.
19. Choose Debug –> Halt.
20. Right-click on the Statistics View area and choose Close from the pop-up menu.

Step B: Use statistics to profile execution.

This step demonstrates creating an STS object to profile the execution of LOG_printf().

1. Double-click on audio.cdb in the Project View to open the Configuration Tool. Right-click on the Statistics Object Manager and choose Insert STS from the pop-up menu. Right-click on

the new object, STS0, and choose Rename from the pop-up menu. Rename the object stsLogPrintf.

2. Choose File –> Save and then close the Configuration Tool.

3. In the Project View, double-click on audio.c to open the file. Because this example also utilizes the Clock module and the Trace module, add trc.h and clk.h to the DSP/BIOS file inclusions after std.h.

```
#include <clk.h>
#include <trc.h>
```

4. Declare the new statistics object by adding the following declaration above main():

```
extern far STS_Obj stsLogPrintf;
```

5. Remove the call to generateSine() from main(). Declare a new integer variable, time, and surround the LOG_printf() function with the following code as shown in the example below:

```
void main()
{
    int date = 25;
    int time;

    BIOS_start();

    if (TRC_query(TRC_USER0) == 0){
        time = CLK_gethtime();
        STS_set(&stsLogPrintf, time);
    }

    LOG_printf(&trace, ”Why do computer scientists celebrate Halloween and
     Christmas on the same day?\
     \nBecause oct %o  equals dec %d.\n”, date, date);


    if (TRC_query(TRC_USER0) == 0){
        time = CLK_gethtime();
        STS_delta(&stsLogPrintf, time);
    }
```

6. Comment out the lines of code for the call to LOG_printf(). This is because our first run with the statistics objects will determine the execution cycles incurred by the instrumentation.

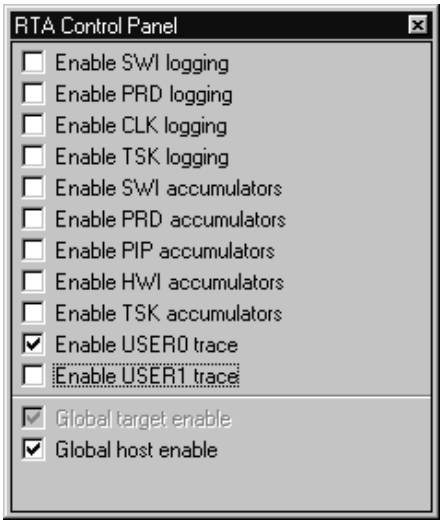
7. Choose File–> Save.

8. Choose Debug –> Reset CPU.

9. Choose Project –> Rebuild All. Load the program.

10. Choose DSP/BIOS –> RTA Control Panel. Right-click on the Control Panel area and deselect Allow Docking. Resize the Control Panel window so that all check boxes are visible.

11. Put check marks in the boxes next to enable USER0 trace and global host enable. Enabling USER0 trace makes the call to TRC_query(TRC_USER0) return 0. Using trace queries can aid in enabling and disabling instrumentation in your programs. As in the example above, if USER0 trace is not enabled in the Control Panel, the calls to CLK_gethtime(), STS_set(), and STS_delta() are not performed.



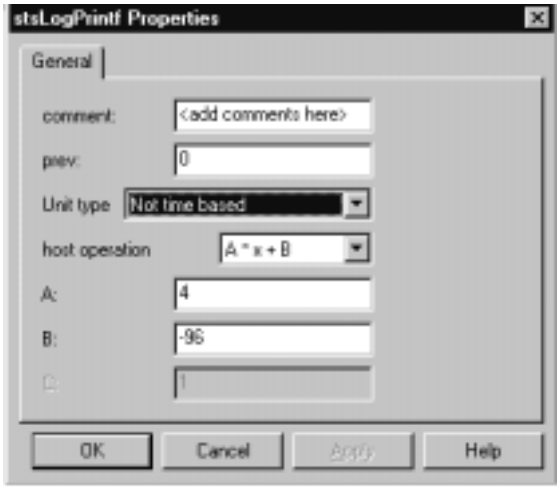
While the statistics and log objects enable you to instrument your code with explicit instrumentation, DSP/BIOS also offers a great deal of implicit instrumentation. There are DSP/BIOS module APIs used internally to collect information about program execution and to provide support for implicit instrumentation. The RTA Control Panel is used to enable various components of this instrumentation. For more information on implicit instrumentation, consult the TMS320C6000 DSP/BIOS User's Guide.

12. Choose DSP/BIOS –> Statistics View. Right-click on the Statistics View area and choose Property Page from the pop-up menu. Select stsLogPrintf from the available statistics. Click OK.

13. Choose Debug –> Go Main and then Debug –> Run.

14. The Statistics View reports an Average and Max value of 30. This value does not represent execution cycles because the statistics object count is incremented only once for every 4 clock ticks. Therefore, you must multiply 30 by 4 in order to determine the number of instruction cycles. So, from the Statistics View information, we can determine that our instrumentation code consumes 120 execution cycles. Choose Debug –> Halt.

15. Return to the Configuration Tool. Right-click on the stsLogPrintf object and choose Properties from the pop-up menu. In the host operation box, choose 'A* x +B.' This enables you to specify certain values for A and B so that value displayed by the Statistics View reflects the multiplication by 4 and also has instrumentation cycles removed. Set A equal to 4, and set B equal to –120. Click OK.

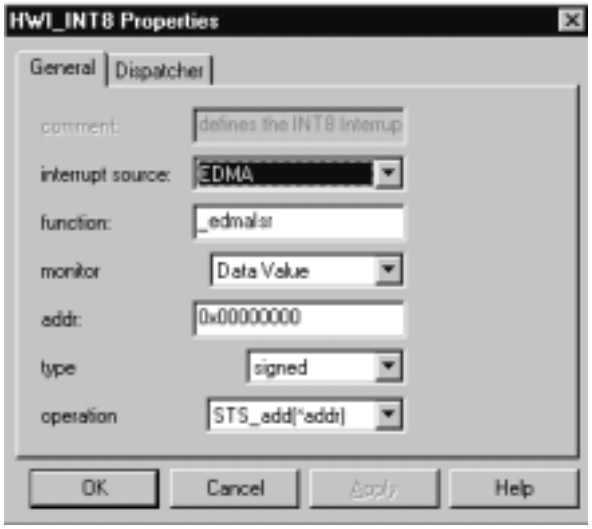


16. Save the configuration file.
17. In audio.c, uncomment the call to LOG_printf(). Choose Project –> Rebuild All. Debug –> Reset CPU. Reload the executable and run. Notice that the call to LOG printf() require only 60 instruction cycles.
18. Choose Debug –> Halt. Right-click on the Statistics View area and choose Close. Right-click on the Control Panel and choose Close.

Step C: Use HWI monitoring to determine interrupts per second.

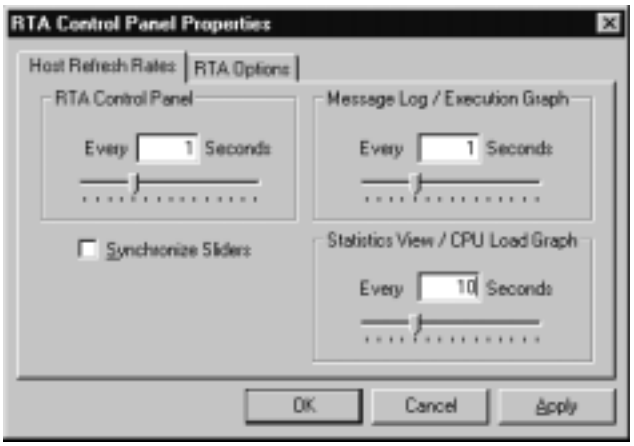
HWI monitoring is enabled through the DSP/BIOS Configuration Tool and no API needs to be added. This is known as implicit instrumentation as opposed to explicit which was described previously in Step B.

1. Double-click audio.cdb file in the Project View. Click on the + sign next to the Hardware Interrupt Service Routine Manager. Right-click on HWI_INT8 and select Properties from the pop-up menu. In the monitor pull-down menu, scroll up and select Data Value. Click OK.



2. Notice that this results in the creation of a new statistics objects, HWI_INT8_STS. Choose File –> Save and close the Configuration Tool.

3. Choose Debug –> Reset CPU.
4. Choose Project –> Rebuild All. Reload the executable.
5. Choose DSP/BIOS –> RTA Control Panel. Put check marks in the boxes next to global host enable and enable HWI accumulators.
6. Choose DSP/BIOS –> Statistics View. Right-click on the Statistics View window and choose Property Page from the pop-up menu. Select the HWI_INT8_STS .
7. Choose Debug –> Go Main and then Debug –> Run. Notice the Count values for the statistics objects. The count is incremented once for every hardware interrupt. In order to determine the number of interrupts per second, find a wrist watch or clock with a second hand. Right-click on the Statistics View and choose Clear from the pop-up menu as you begin monitoring the time on the watch. Selecting clear refreshes the statistics values to zero. Wait for ten seconds and then examine the Count value. Divide Count by ten to determine the number of interrupts per second.
8. Alternatively, you can have the Statistics View update only once every ten seconds. Then you can take the initial count value and divide by ten to determine the interrupts per second. To do so, right-click on the RTA Control Panel. Select Property Page from the pop-up menu. This displays the host refresh rates for different tools. Change the Statistics View / CPU Load Graph refresh rate to 10 seconds. Click OK.



9. Right-click on the Statistics View and select Clear. Take note of the first count value and divide by ten to determine interrupts per second.
10. Choose Debug –> Halt. Close the RTA Control Panel and Statistics View by right-clicking and selecting Close.

5 The Third Degree: Going All The Way

5.1 Using the DSP/BIOS Scheduler

The third degree demonstrates how to migrate over to the DSP/BIOS scheduler. This stage shows how DSP/BIOS departs from traditional main loop programming by executing the application threads within the priority-based scheduler rather than inside of the main() function. Because the scheduler is priority-based, it provides an effective solution to those classes of real time programs that require preemption (two threads that execute at different periodic rates, multi-rate signal processing, multi-channel applications, etc.). A general guideline for setting the priority of threads is to set the priority higher on 'smaller' more frequently executed threads to allow them to preempt 'bigger' longer running, less frequently executed threads.

To transfer the processing from main() to the scheduler, you incorporate the functions from the executive loop into the Configuration Tool and classify the functions as DSP/BIOS threads. DSP/BIOS offers a variety of thread types with differing priority levels: hardware interrupts, software interrupts, tasks, clock objects, periodic functions, and idle functions. In this example, software interrupts (SWIs) are created to perform the work previously done in main(). The SWI module is used to create and configure the SWIs. Software interrupts are modeled after hardware interrupt service routines. They run to completion, and are prioritized with fifteen priority levels (priority one is the lowest, and priority fifteen is the highest). All software interrupts preempt tasks and the background idle loop while hardware interrupts preempt software interrupts. To program using software interrupts, use the event-driven programming model. For event handling that is too intensive for a hardware interrupt, assign a software interrupt to defer processing when it is appropriate. For example, a hardware interrupt puts data into a buffer; when the buffer is full, a software interrupt is posted and processing on the buffer occurs. This allows the programmer to assign various levels of priorities to not only hardware events, but also to software events in order to fully utilize the CPU's capabilities.

It is the scheduler's job to manage the execution of the DSP/BIOS threads. For instance, the scheduler runs after calling the DSP/BIOS interrupt handlers (either HWI Dispatcher or the HWI_enter /exit macros) ,a thread makes a DSP/BIOS scheduling call (such as SWI_post() or SEM_post) or a DSP/BIOS thread finishes execution. When a software interrupt is posted using the SWI_post() API, the SWI object's function is not necessarily executed immediately. Instead, the function is scheduled for execution. When the scheduler runs, it checks the priorities of posted SWIs to determine which to run next, or whether to preempt the thread currently running. When no threads are pending, the idle loop runs through the idle functions configured under the IDL Manager.

Step A: Remove the processing from main().

The first step is to remove the majority of the processing from main() so that they can be incorporated into SWI thread objects by the Configuration Tool, and thereby be managed by the scheduler.

1. In C:\ti\myprojects\biosbydegrees, create a new folder called Degree3. Copy all the files from Degree2 into Degree3.
2. Double-click on the Code Composer Studio icon on the desktop. Choose Project –> Open and browse to the Degree3 folder. Select the audio.pjt file to open the project.
3. In the Project View, click on the + sign next to the Project folder and then next to audio.pjt to expand the project. Click on the + sign next to the Source folder and the DSP/BIOS Config folder.

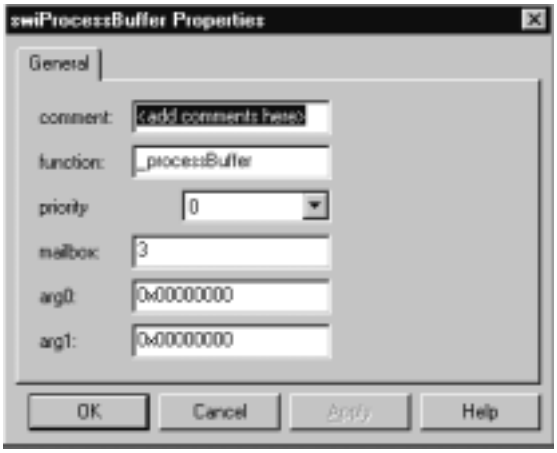
4. Double-click on audio.c to open the source file.
5. Reduce the code in main() so that it looks like the following example. The call to BIOS_start() is no longer necessary because it runs automatically when the program returns from main().

```
/*
 * ======== main ========
 */
void main()
{
    int date = 25;

    CSL_init();                /* Initialize CSL                        */

    initApplication();          /* Initialize Peripherals               */
    interruptsEnable();         /* Enable EDMA and Global Interrupts    */

    LOG_printf(&trace, "Why do computer scientists celebrate Halloween and
      Christmas on the same day?\
      \nBecause oct %o  equals dec %d.\n", date, date);


    /* Fall into DSP/BIOS idle loop */
    return;
}
```

Step B: Turn processBuffer() into a software interrupt handler.

processBuffer() is no longer called in main(). Now that you are using the scheduler, processBuffer() runs as a result of a software interrupt. When the EDMA finishes transferring a frame of data into memory, its interrupt service routine posts a software interrupt which causes processBuffer() to run.

1. Double-click on audio.cdb. In the Configuration Tool, right-click on the Software Interrupt Manager and choose Insert SWI from the pop-up menu. This creates a new SWI object called SWI0.
2. Right-click on SWI0 and choose Rename from the pop-up menu. Rename the object swiProcessBuffer. Right-click on swiProcessBuffer and choose Properties from the pop-up menu. In the function box, replace FXN_F_nop with _processBuffer. Remember to precede processBuffer with an underscore. In the mailbox box, type 3. Click OK.



Step C: Modify edmaIsr to post the software interrupt.

Now that processBuffer() runs as result of a software interrupt, the interrupt service routine edmaIsr, must post the software interrupt when the inputReadyFlag or outputReadyFlag are set.

1. In order to post the software interrupt for swiProcessBuffer, you must include the SWI module header file. Double-click on device.c in the Project View. Add the following lines to device.c:

   ```
   #include <std.h>
   #include <swi.h>
   ```

   Remember that header files for DSP/BIOS modules must be included after std.h.

2. Declare the SWI object within device.c. Add the following declaration after the file inclusions:

   ```
   /* Object created by the Configuration Tool */
   extern far SWI_Obj swiProcessBuffer;
   ```

3. Modify edmaIsr() in device.c as in the example below.

   ```
   void edmaIsr(void)
   {
       int  edmaChannel = 0;
       edmaChannel = EDMA_RGET(CIPR);
       if (edmaChannel & 0x0010){          /* Action= Ping_RX full     */
             EDMA_RSET(CIPR, 0x0010);      /* Reset Pending Flag       */
             EdmaBufferMask |= RXPINGMASK;
             inputReadyFlag = 1;
             SWI_andn(&swiProcessBuffer, 0x001);
       }
       else if(edmaChannel & 0x0020){      /* Action= Pong_RX full     */
             EDMA_RSET(CIPR, 0x0020);      /* Reset Pending Flag       */
             edmaBufferMask |= RXPONGMASK;
             inputReadyFlag = 1;
             SWI_andn(&swiProcessBuffer, 0x001);
       }
       else if (edmaChannel & 0x0040){     /* Action= Pong_TX full    */
             EDMA_RSET(CIPR, 0x0040);      /* Reset Pending Flag      */
             edmaBufferMask |= TXPONGMASK;
             outputReadyFlag = 1;
             SWI_andn(&swiProcessBuffer, 0x02);
       }
       else if (edmaChannel & 0x0080){     /* Action= Ping_TX full    */
             EDMA_RSET(CIPR, 0x0080);      /* Reset Pending Flag      */
             edmaBufferMask |= TXPINGMASK;
             outputReadyFlag = 1;
             SWI_andn(&swiProcessBuffer, 0x02);
   ```

```
        }
        else {
              EDMA_RSET(CIPR, edmaChannel);      /* Reset Pending Flag      */
        }
    }
```

4. In the next step you use HWI dispatcher to handle an interrupt. When an HWI object does not use the dispatcher, the HWI_enter assembly macro must be called prior to any DSP/BIOS API calls that could post or affect a software interrupt, and the HWI_exit assembly macro must be called at the very end of the function's code (for more detail on HWI_enter/HWI_exit macros, refer to the *DSP/BIOS API Reference Guide*).

   WARNING: Remove the word "interrupt" from the edmaIsr function definition in the device.c source file.

5. Choose File –> Save. Close the file.

Step D: Alter the configuration file so that the HWI dispatcher handles the hardware interrupts.

1. In the Configuration Tool, click on the + sign next to the Hardware Interrupt Service Routine Manager. Right-click on HWI_INT8 and choose Properties from the pop-up menu. Put a check mark in the Use Dispatcher box. Click OK.

2. Choose File –> Save. Close the Configuration Tool by choosing File –> Close.

Step E: Run the program.

1. Choose Project –> Rebuild All. Choose Debug –> Reset CPU. Choose File –> Load Program and load the executable audio.out. Choose Debug –> Go Main and then Debug –> Run.

Appendix A Source Code Listings for Zero Degree – non-BIOS Audio Application

```
– – – – – – – –  File: audio.c – – – – – – – – – – – – – – –

/*
 *  ======== audio.c ========
 */

#include <stdio.h>
#include <csl.h>
#include <csl_edma.h>
#include <csl_irq.h>
#include <csl_mcbsp.h>
#include <fir_gen.h>
#include ”audio.h”

volatile int inputReadyFlag = 0;  /* Set when RX buffer is full */
volatile int outputReadyFlag = 0; /* Set when TX buffer is ready for new data */
volatile int edmaBufferMask = 0;  /* Mask indicates which buffers are ready   */

volatile int filterEnable = 0;    /* Change to 1 to enable filtering */
volatile int filterSelect = 5;    /* 0–8 selects one of nine filter coefficients */

MCBSP_Handle hMcbsp;
EDMA_Handle hEdmaTx;
EDMA_Handle hEdmaRx;

/* Global Variables */
short pingRX[BUFSIZE];
short pongRX[BUFSIZE];
short pingTX[BUFSIZE];
short pongTX[BUFSIZE];

/* Function Prototypes */
static void processBuffer(void);

/* External References */
extern short filterCoeff[FILTERS][COEFFS];

/*
 *  ======== main ========
 */
void main(void)
{
    int date = 25;

    CSL_init();                /* Initialize CSL                       */

    initApplication();         /* Initialize Peripherals               */
    interruptsEnable();        /* Enable EDMA and Global Interrupts    */

    printf(”Why do computer scientists celebrate Halloween and Christmas \
     on the same day?  Because oct %o equals dec %d.\n”, date, date);

    while (1) {
        if (inputReadyFlag && outputReadyFlag) {
            inputReadyFlag = 0;       /* Reset flags      */
            outputReadyFlag = 0;
            processBuffer();          /* Process Data     */
        }
    }
}
```

```
/*
 *  ======== processBuffer ========
 */
static void processBuffer(void)
{
    short *processSrc;
    short *processDst;
    int gie;
    int i;

   /*
    *  Disable interrupts to avoid possible
    *  race condition with edmaISR which also
    *  changes edmaBufferMask
    */

    gie = IRQ_globalDisable();

    if (edmaBufferMask & RXPINGMASK) {
        processSrc = (short *)&pingRX[0];          /* Set-up Source    */

        /* Reprogram EDMA with pongRX as destination */
        EDMA_configArgs(
          hEdmaRx,
          EDMA_OPT_RMK(
            EDMA_OPT_PRI_HIGH,
            EDMA_OPT_ESIZE_16BIT,
            EDMA_OPT_2DS_NO,
            EDMA_OPT_SUM_NONE,
            EDMA_OPT_2DD_NO,
            EDMA_OPT_DUM_INC,
            EDMA_OPT_TCINT_YES,
            EDMA_OPT_TCC_OF(0X5),
            EDMA_OPT_LINK_NO,
            EDMA_OPT_FS_NO),
          MCBSP_getRcvAddr(hMcbsp),
          PRESENT,
          (Uint32)&pongRX[HISTORY],
          EDMA_IDX_DEFAULT,
          EDMA_RLD_DEFAULT
        );

       /*
        *  Program QDMA to transfer data at end of full buffer to
        *  beginning of next buffer for filtering purposes
        */

        EDMA_qdmaConfigArgs(
          EDMA_OPT_RMK(
            EDMA_OPT_PRI_LOW,
            EDMA_OPT_ESIZE_16BIT,
            EDMA_OPT_2DS_NO,
            EDMA_OPT_SUM_INC,
            EDMA_OPT_2DD_NO,
            EDMA_OPT_DUM_INC,
            EDMA_OPT_TCINT_NO,
            EDMA_OPT_TCC_OF(0X0),
            EDMA_OPT_LINK_NO,
            EDMA_OPT_FS_NO),
          (Uint32)&pingRX[PRESENT],
          HISTORY,
          (Uint32)&pongRX[0],
```

```
      EDMA_IDX_DEFAULT
    );

}
else if (edmaBufferMask & RXPONGMASK) {
    processSrc = (short *)&pongRX[0];

    /* Reprogram EDMA with pingRX as destination */
    EDMA_configArgs(
      hEdmaRx,
      EDMA_OPT_RMK(
        EDMA_OPT_PRI_HIGH,
        EDMA_OPT_ESIZE_16BIT,
        EDMA_OPT_2DS_NO,
        EDMA_OPT_SUM_NONE,
        EDMA_OPT_2DD_NO,
        EDMA_OPT_DUM_INC,
        EDMA_OPT_TCINT_YES,
        EDMA_OPT_TCC_OF(0X4),
        EDMA_OPT_LINK_NO,
        EDMA_OPT_FS_NO),
      MCBSP_getRcvAddr(hMcbsp),
      PRESENT,
      (Uint32)&pingRX[HISTORY],
      EDMA_IDX_DEFAULT,
      EDMA_RLD_DEFAULT
    );

   /*
    *  Program QDMA to transfer data at end of full buffer to
    *  beginning of next buffer for filtering purposes
    */

    EDMA_qdmaConfigArgs(
      EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,
        EDMA_OPT_ESIZE_16BIT,
        EDMA_OPT_2DS_NO,
        EDMA_OPT_SUM_INC,
        EDMA_OPT_2DD_NO,
        EDMA_OPT_DUM_INC,
        EDMA_OPT_TCINT_NO,
        EDMA_OPT_TCC_OF(0X0),
        EDMA_OPT_LINK_NO,
        EDMA_OPT_FS_NO),
      (Uint32)&pongRX[PRESENT],
      HISTORY,
      (Uint32)&pingRX[0],
      EDMA_IDX_DEFAULT
    );
}
if (edmaBufferMask & TXPONGMASK) {
    processDst = (short *)&pongTX[HISTORY];    /* Set-up Dest      */

    /* Reprogram EDMA with pingTX as source */
    EDMA_configArgs(
      hEdmaTx,
      EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,
        EDMA_OPT_ESIZE_16BIT,
        EDMA_OPT_2DS_NO,
        EDMA_OPT_SUM_INC,
        EDMA_OPT_2DD_NO,
```

```
            EDMA_OPT_DUM_NONE,
            EDMA_OPT_TCINT_YES,
            EDMA_OPT_TCC_OF(0X7),
            EDMA_OPT_LINK_NO,
            EDMA_OPT_FS_NO),
          (Uint32)&pingTX[HISTORY],
          PRESENT,
          MCBSP_getXmtAddr(hMcbsp),
          EDMA_IDX_DEFAULT,
          EDMA_RLD_DEFAULT
        );
    }
    else if (edmaBufferMask & TXPINGMASK) {
        processDst = (short *)&pingTX[HISTORY];

        /* Reprogram EDMA with pongTX as source */
        EDMA_configArgs(
          hEdmaTx,
          EDMA_OPT_RMK(
            EDMA_OPT_PRI_LOW,
            EDMA_OPT_ESIZE_16BIT,
            EDMA_OPT_2DS_NO,
            EDMA_OPT_SUM_INC,
            EDMA_OPT_2DD_NO,
            EDMA_OPT_DUM_NONE,
            EDMA_OPT_TCINT_YES,
            EDMA_OPT_TCC_OF(0X6),
            EDMA_OPT_LINK_NO,
            EDMA_OPT_FS_NO),
         (Uint32)&pongTX[HISTORY],
         PRESENT,
         MCBSP_getXmtAddr(hMcbsp),
         EDMA_IDX_DEFAULT,
         EDMA_RLD_DEFAULT
       );
    }

    edmaBufferMask = 0;                 /* Clear ISR edma flag          */

    IRQ_globalRestore(gie);             /* Restore global interrupts    */

    if (filterEnable == 0) {            /* Loop-Back if filter off      */
        for (i = 0; i < PRESENT; i++) {
            processDst[i] = processSrc[i];
        }
    }
    else {                              /* Apply filter function        */
        fir_gen(processSrc, (short *)&filterCoeff[filterSelect][0], processDst,
           COEFFS, PRESENT);
    }

    /* 0 least significant bit for transfer to codec */
    for ( i = 0; i < PRESENT; i++ ) {
        processDst[i] &= 0xfffe;
    }
}
```

```
– – – – – – – –  File: device.c – – – – – – – – – – – – – – –
/*
 *  ======== device.c ========
 */
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include ”audio.h”

/* Function Prototypes */
void initApplication(void);
void interruptsEnable(void);
static void edmaEnable(void);
static void mcbspEnable(void);
static void codecEnable(void);
static void mcbspWrite(unsigned int outdata);
static unsigned int mcbspRead(void);
static unsigned int codecReadControl(unsigned int reg);
static void codecWriteControl(unsigned int reg, unsigned int data);
static void codecError(int id);

/*
 *  Warning: Remove ”interrupt” keyword (below) when making DSP/BIOS
 *  scheduled calls from within an interrupt.
 */

interrupt void edmaIsr(void);

/* External References */
extern volatile int inputReadyFlag;
extern volatile int outputReadyFlag;
extern volatile int edmaBufferMask;

/*
 *  ======== initApplication ========
 */
void initApplication(void)
{
    int i;

    for (i = 0; i < BUFSIZE; i++) {
        pingRX[i] = 0x0000;
        pongRX[i] = 0x0000;
        pingTX[i] = 0x0000;
        pongTX[i] = 0x0000;
    }

    mcbspEnable();                  /* Configure McBSP0 */
    codecEnable();                  /* Configure Codec */
    edmaEnable();                   /* Enable EDMA */
}

/*
 *  ======== interruptsEnable ========
 */
void interruptsEnable(void)
{
    IER |= 0x00000002;              /* enable NMIE */
    IRQ_globalRestore(1);           /* enable GIE  */
}

/*
 *  ======== mcbspEnable ========
```

```
 */
static void mcbspEnable(void)
{
    /* Needed to set free bit */
    volatile int *spcr0 = (volatile int *)MCBSP_ADDR(SPCR0);

    /* Get a handle for McBSP0 */
    hMcbsp = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);

    /* Configure the Serial Port */
    MCBSP_configArgs(
        hMcbsp,
        MCBSP_SPCR_RMK(
         MCBSP_SPCR_FREE_DEFAULT,
         MCBSP_SPCR_SOFT_DEFAULT,
            MCBSP_SPCR_FRST_YES,
            MCBSP_SPCR_GRST_YES,
            MCBSP_SPCR_XINTM_XRDY,
            MCBSP_SPCR_XSYNCERR_NO,
            MCBSP_SPCR_XRST_NO,
            MCBSP_SPCR_DLB_OFF,
            MCBSP_SPCR_RJUST_RZF,
            MCBSP_SPCR_CLKSTP_DISABLE,
            MCBSP_SPCR_DXENA_OFF,
            MCBSP_SPCR_RINTM_RRDY,
            MCBSP_SPCR_RSYNCERR_NO,
            MCBSP_SPCR_RRST_NO
            ),
        MCBSP_RCR_RMK(
            MCBSP_RCR_RPHASE_SINGLE,
            MCBSP_RCR_RFRLEN2_OF(0),
            MCBSP_RCR_RWDLEN2_8BIT,    /* Not using Phase */
            MCBSP_RCR_RCOMPAND_MSB,
            MCBSP_RCR_RFIG_YES,
            MCBSP_RCR_RDATDLY_1BIT,
            MCBSP_RCR_RFRLEN1_OF(0),
            MCBSP_RCR_RWDLEN1_16BIT,
            MCBSP_RCR_RWDREVRS_DISABLE),
        MCBSP_XCR_RMK(
            MCBSP_XCR_XPHASE_SINGLE,
            MCBSP_XCR_XFRLEN2_OF(0),
            MCBSP_XCR_XWDLEN2_8BIT,    /* Not using Phase */
            MCBSP_XCR_XCOMPAND_MSB,
            MCBSP_XCR_XFIG_YES,
            MCBSP_XCR_XDATDLY_1BIT,
            MCBSP_XCR_XFRLEN1_OF(0),
            MCBSP_XCR_XWDLEN1_16BIT,
            MCBSP_XCR_XWDREVRS_DISABLE),
        MCBSP_SRGR_DEFAULT,
        MCBSP_MCR_DEFAULT,
        MCBSP_RCER_DEFAULT,
        MCBSP_XCER_DEFAULT,
        MCBSP_PCR_DEFAULT);

    *spcr0 |= 0x02000000;           /* Sets Free Bit */
}

/*
 *  ======== mcbspWrite ========
 */
static void mcbspWrite(unsigned int outdata)
{
    while (!(MCBSP_xrdy(hMcbsp))) {
```

```
    }

    MCBSP_write(hMcbsp, outdata);
}

/*
 *  ======== mcbspRead ========
 */
static unsigned int mcbspRead(void)
{
    while (!(MCBSP_rrdy(hMcbsp))) {
    }

    return (MCBSP_read(hMcbsp));
}

/*
 *  ======== codecEnable ========
 */
static void codecEnable(void)
{
    unsigned int temp;

   /*
    *  Perform Voice Channel Initialization of TLC320AD535 Codec
    *  AD535 has 2 serial port channels – Data, Voice
    *  Data  Channel Controlled by Registers 1,2    (Reg 0 = NOP)
    *  Voice Channel Controlled by Registers 3,4,5,6
    *  Only Voice channel used on DSK 6211
    */

    /* Set–Up Register 0 (NOP) – Dummy Read/Write Codec */
    codecWriteControl(0, 0);
    temp = codecReadControl(0);

   /*
    *  Set–Up Register 1 / 2   – Only Used by Data Serial Port –NA
    *  Set–Up Register 3       – S/W Reset + Power Down + No loop / gain=0dB
    */

    codecWriteControl(3, 0x00C6);       /* + With Reset    */
    codecWriteControl(3, 0x0006);       /* + Without Reset */
    temp = codecReadControl(3);
    if ((temp & 0x00ff) != 0x0006) {
        codecError(3);
    }

    /* Set–Up Register 4       – Voice ADC gain = 0dB   */
    codecWriteControl(4, 0x0000);
    temp = codecReadControl(4);
    if ((temp & 0x00ff) != 0x0000) {
        codecError(4);
    }

    /* Set–Up Register 5       – Spkr L/R gain = 0dB    */
    codecWriteControl(5, 0x0002);
    temp = codecReadControl(5);
    if ((temp & 0x00fe) != 0x0002) {
        codecError(5);
    }

    /* Set–Up Register 6       – Handset gain  = 0dB    */
    codecWriteControl(6, 0x0000);
```

```
    temp = codecReadControl(6);
    if ((temp & 0x0080) != 0x0000) {
        codecError(6);
    }

    /* Set–Up Register – (NOP)      */
    codecWriteControl(0, 0);
    temp = codecReadControl(0);
}

/*
 *  ======== codecReadControl ========
 */
static unsigned int codecReadControl(unsigned int reg)
{
    unsigned int temp = 0;

    temp = (((reg & 0x001F) << 8) | 0x2000);

    mcbspWrite(0);
    mcbspRead( );

    mcbspWrite(1);
    mcbspRead( );

    mcbspWrite(temp);
    temp = mcbspRead( );

    mcbspWrite(0);
    mcbspRead( );

    return temp;
}

/*
 *  ======== codecWriteControl ========
 */
static void codecWriteControl(unsigned int reg, unsigned int data)
{
    unsigned int temp = 0;

    temp = ((reg & 0x001F) << 8) | (data & 0x00ff);

    mcbspWrite(0);
    mcbspRead( );

    mcbspWrite(1);
    mcbspRead( );

    mcbspWrite(temp);
    mcbspRead( );

    mcbspWrite(0);
    mcbspRead( );
}

/*
 *  ======== edmaEnable ========
 */
static void edmaEnable(void)
{
    /* General EDMA Initialization  */
    EDMA_reset(EDMA_HINV);       /* Clear all pending events          */
```

```
    EDMA_RSET(CIER, 0x000);
    EDMA_RSET(CIPR, 0xFFFF);     /* Clear all pending Queued EDMA ints */

    hEdmaTx = EDMA_open(EDMA_CHA_XEVT0, 0);              /* Channel 12 */
    hEdmaRx = EDMA_open(EDMA_CHA_REVT0, 0);              /* Channel 13 */

    /* Initialize EDMA for Transfer from McBSP Channel 0 */
    EDMA_configArgs(
      hEdmaRx,
      EDMA_OPT_RMK(
        EDMA_OPT_PRI_HIGH,
        EDMA_OPT_ESIZE_16BIT,
        EDMA_OPT_2DS_NO,
        EDMA_OPT_SUM_NONE,
        EDMA_OPT_2DD_NO,
        EDMA_OPT_DUM_INC,
        EDMA_OPT_TCINT_YES,
        EDMA_OPT_TCC_OF(0X4),
        EDMA_OPT_LINK_NO,
        EDMA_OPT_FS_NO),
      MCBSP_getRcvAddr(hMcbsp),
      PRESENT,
      (Uint32)&pingRX[HISTORY],
      EDMA_IDX_DEFAULT,
      EDMA_RLD_DEFAULT
    );

    /* Initialize EDMA for Transfer Samples back to McBSP Channel 0 */
    EDMA_configArgs(
      hEdmaTx,
      EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,
        EDMA_OPT_ESIZE_16BIT,
        EDMA_OPT_2DS_NO,
        EDMA_OPT_SUM_INC,
        EDMA_OPT_2DD_NO,
        EDMA_OPT_DUM_NONE,
        EDMA_OPT_TCINT_YES,
        EDMA_OPT_TCC_OF(0X6),
        EDMA_OPT_LINK_NO,
        EDMA_OPT_FS_NO),
      (Uint32)&pongTX[HISTORY],
      PRESENT,
      MCBSP_getXmtAddr(hMcbsp),
      EDMA_IDX_DEFAULT,
      EDMA_RLD_DEFAULT
    );

    /* Enable Channels */
    EDMA_enableChannel(hEdmaTx);
    EDMA_enableChannel(hEdmaRx);

    /* Enable Rx/Tx Complete Interrupts */
    EDMA_RSET(CIER, 0x00f0);

    /* Enable Interrupts IRQ's  */
    IRQ_map(IRQ_EVT_EDMAINT, 8);    /* map DMA event to Hardware Int 8      */
    IRQ_enable(IRQ_EVT_EDMAINT);    /* Enable EDMA-to-CPU interrupt (int 8) */
}

/*
 *  ======== codecError ========
 */
```

```
static void codecError (int id)
{
    /* Codec Initialization Error – Exit and Reset DSK */
    for (;;) {
    }       /* loop forever */
}

/*
 *  ======== edmaIsr ========
 */
interrupt void edmaIsr(void)
{

   /*
    *  This ISR is used to provide signalling to processing code.
    *  This ISR is called under 4 conditions:
    *  1. pingRX Buffer has completed filling.        CIPR = 10h
    *  2. pongRX Buffer has completed filling.        CIPR = 20h
    *  3. pongTX Buffer has been transmitted.         CIPR = 40h
    *  4. pingTX Buffer has been transmitted.         CIPR = 80h
    *  CIPR is read to determine the cause of the interrupt.
    */

    int edmaChannel = 0;

    edmaChannel = EDMA_RGET(CIPR);

    if (edmaChannel & 0x0010) {         /* pingRX full          */
        EDMA_RSET(CIPR, 0x0010);        /* Reset Pending Flag   */
        edmaBufferMask |= RXPINGMASK;
        inputReadyFlag = 1;
    }
    else if (edmaChannel & 0x0020) {    /* pongRX full          */
        EDMA_RSET(CIPR, 0x0020);        /* Reset Pending Flag   */
        edmaBufferMask |= RXPONGMASK;
        inputReadyFlag = 1;
    }
    else if (edmaChannel & 0x0040) {    /* pongTX empty        */

        EDMA_RSET(CIPR, 0x0040);        /* Reset Pending Flag   */
        edmaBufferMask |= TXPONGMASK;
        outputReadyFlag = 1;
    }
    else if (edmaChannel & 0x0080) {    /* pingTX empty         */
        EDMA_RSET(CIPR, 0x0080);        /* Reset Pending Flag   */
        edmaBufferMask |= TXPINGMASK;
        outputReadyFlag = 1;
    }
    else {
        EDMA_RSET(CIPR, edmaChannel);   /* Reset Pending Flag   */
    }
}
```

```
– – – – – – – –  File: audio.h – – – – – – – – – – – – – – –

/*
 *  ======== audio.h ========
 */

/*
 *  For filtering purposes, number of samples
 *  that are copied from the end of one full
 *  buffer to the beginning of the next buffer
 *  to be filled
 */
#include <csl_mcbsp.h>
#include <csl_edma.h>

#define HISTORY             32
#define PRESENT             1000    /* Number of new samples put into buffer */
#define BUFSIZE             (HISTORY + PRESENT)

#define FILTERS             9
#define COEFFS              HISTORY

#define PI 3.1415927

#define RXPINGMASK      0x0001
#define RXPONGMASK      0x0010
#define TXPINGMASK      0x0100
#define TXPONGMASK      0x1000

/* Global Variables */
extern short pingRX[BUFSIZE];
extern short pongRX[BUFSIZE];
extern short pingTX[BUFSIZE];
extern short pongTX[BUFSIZE];

extern MCBSP_Handle hMcbsp;
extern EDMA_Handle hEdmaTx;
extern EDMA_Handle hEdmaRx;
```

```
– – – – – – – –  File: vectors.asm – – – – – – – – – – – – – – –
;
; vectors.asm
;

            .ref    _c_int00
            .ref    _edmaIsr

; Macro Definition
; For unused entries in Int Vector Table

unused      .macro  id
            .global unused:id:
unused:id:
            b       unused:id:
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            .endm

; Interrupt Vector Table

            .sect         ”vectors”         ; Define Memory Section

_RESET:     mvkl    _c_int00, b0            ; Reset Vector
            mvkh    _c_int00, b0
            b       b0
            nop
            nop
            nop
            nop
            nop

_NMI:       unused  NMI
_RESV1:     unused  RESV1
_RESV2:     unused  RESV2
_INT4:      unused  4
_INT5:      unused  5
_INT6:      unused  6
_INT7:      unused  7

_INT8:      mvkl    _edmaIsr, b0            ; EDMA Service Function
            mvkh    _edmaIsr, b0
            b       b0
            nop
            nop
            nop
            nop
            nop

_INT9:      unused  9
_INT10:     unused  10
_INT11:     unused  11
_INT12:     unused  12
_INT13:     unused  13
_INT14:     unused  14
_INT15:     unused  15

            .end
```

```
- - - - - - - -  File: linker.cmd - - - - - - - - - - - - - - -
/*
 * ======== linker.cmd ========
 */
MEMORY
{
   VECS:       o = 00000000h   l = 00000200h
   IRAM:       o = 00000200h   l = 0000FE00h
}
SECTIONS
{
   vectors     >       VECS
   .cinit      >       IRAM
   .text       >       IRAM
   .stack      >       IRAM
   .bss        >       IRAM
   .const      >       IRAM
   .data       >       IRAM
   .far        >       IRAM
   .switch     >       IRAM
   .sysmem     >       IRAM
   .cio        >       IRAM
}
```

```
--------------File: coeffs.c ------------------------------
/*
 *  ======== coeffs.c ========
 */
#include ”audio.h”

/* ten low pass filter coefficient sets */

const short filterCoeff[FILTERS][COEFFS] = {

/* 0 8kHz, 32 Taps, 100 Hz, 400Hz, pass ripple 1.348dB, stop atten 37.391dB  */
0x0145,0x00f7,0x014b,0x01ab,0x0214,0x0286,0x02fc,0x0375,
0x03ec,0x045e,0x04c9,0x0528,0x0578,0x05b6,0x05e1,0x05f7,
0x05f7,0x05e1,0x05b6,0x0578,0x0528,0x04c9,0x045e,0x03ec,
0x0375,0x02fc,0x0286,0x0214,0x01ab,0x014b,0x00f7,0x0145,    // 100 – 400Hz

/* 1 8kHz, 32 Taps, 400 Hz, 800Hz, pass ripple 1.527dB, stop atten 45.815dB  */
0xffd0,0xff04,0xfeaa,0xfdf6,0xfd6a,0xfd1d,0xfd48,0xfe13,
0xff9a,0x01db,0x04b8,0x07f1,0x0b31,0x0e15,0x1041,0x116b,
0x116b,0x1041,0x0e15,0x0b31,0x07f1,0x04b8,0x01db,0xff9a,
0xfe13,0xfd48,0xfd1d,0xfd6a,0xfdf6,0xfeaa,0xff04,0xffd0,    // 400 – 800Hz

/* 2 8kHz, 32 Taps, 800 Hz,1300Hz, pass ripple 0.897dB, stop atten 51.332dB  */
0x0015,0x0043,0x0085,0x00AD,0x006C,0xFF69,0xFD7D,0xFAEC,
0xF88C,0xF79B,0xF957,0xFE63,0x0654,0x0F94,0x17C7,0x1C9A,
0x1C9A,0x17C7,0x0F94,0x0654,0xFE63,0xF957,0xF79B,0xF88C,
0xFAEC,0xFD7D,0xFF69,0x006C,0x00AD,0x0085,0x0043,0x0015,

/* 3 8kHz, 32 Taps, 1200Hz,1800Hz, pass ripple 0.428dB, stop atten 56.833dB  */
0xffdc,0x0030,0x00f0,0x0171,0x00a4,0xfeb1,0xfda4,0xff7f,
0x032a,0x0440,0xff98,0xf887,0xf817,0x04d3,0x1a98,0x2ba3,
0x2ba3,0x1a98,0x04d3,0xf817,0xf887,0xff98,0x0440,0x032a,
0xff7f,0xfda4,0xfeb1,0x00a4,0x0171,0x00f0,0x0030,0xffdc,    // 1200 – 1800Hz

/* 4 8kHz, 32 Taps, 1700Hz,2400Hz, pass ripple 0.192dB, stop atten 63.755dB  */
0xffd6,0x0017,0x00d9,0x009e,0xfefc,0xff04,0x01ba,0x01ca,
0xfd42,0xfcf0,0x046c,0x0557,0xf83f,0xf52d,0x131c,0x3951,
0x3951,0x131c,0xf52d,0xf83f,0x0557,0x046c,0xfcf0,0xfd42,
0x01ca,0x0aba,0xff04,0xfefc,0x009e,0x00d9,0x0017,0xffd6,    // 1700 – 2400Hz

/* 5 8kHz, 32 Taps, 2200Hz,2900Hz, pass ripple 0.179dB, stop atten 64.375dB  */
0xffdb,0x006d,0x00d4,0xff58,0xff7d,0x01b3,0xff53,0xfdcd,
0x0328,0x00a3,0xfa30,0x0497,0x05b5,0xf0a4,0x057c,0x437c,
0x437c,0x057c,0xf0a4,0x05b5,0x049c,0xfa30,0x00a3,0x0328,
0xfdcd,0xff53,0x01b3,0xff7d,0xff58,0x00d4,0x006d,0xffdb,    // 2200 – 2900Hz

/* 6 8kHz, 32 Taps, 2800Hz,3600Hz, pass ripple 0.087dB, stop atten 70.714dB  */
0xffcf,0xffa6,0x0088,0xff7f,0x0004,0x00ed,0xfe16,0x023e,
0xfec7,0xfea6,0x04d3,0xf861,0x077f,0xfe34,0xf1dc,0x4cbe,
0x4cbe,0xf1dc,0xfe34,0x077f,0xf861,0x04d3,0xfea6,0xfec7,
0x023e,0xfe16,0x00ed,0x0004,0xff7f,0x0088,0xffa6,0xffcf,    // 2800 – 3600Hz
```

```
/* 7 8kHz, 32 Taps, 3300Hz,3900Hz, pass ripple 0.286dB, stop atten 60.307dB  */
0xfeb0,0x007b,0xffb4,0xffe3,0x00c1,0xfe6e,0x0271,0xfccb,
0x03a8,0xfc75,0x0297,0xff85,0xfcbc,0x09c0,0xe902,0x500e,
0x500e,0xe902,0x09c0,0xfcbc,0xff85,0x0297,0xfc75,0x03a8,
0xfccb,0x0271,0xfe6e,0x00c1,0xffe3,0xffb4,0x007b,0xfeb0,    //3300 – 3900Hz

/* 8 No Filter                                                              */
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x7fff     // None!
};
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265