

# **DSP/BIOS by Degrees: Using DSP/BIOS (Code Composer Studio v1.2) in an Existing Application**

---

*Thomas Maughan, Kathryn Rafac*

*Texas Instruments Incorporated*

## **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 by providing a priority-based scheduler, a set of DSP/BIOS real-time analysis (RTA) tools, and real-time data exchange (RTDX™) between the host computer and the DSP.

The benefits of real-time analysis provided by DSP/BIOS are often required in programs that were engineered without it. When the program is not built from the ground up using the DSP/BIOS kernel and real-time analysis features, the lack of familiarity prevents engineers from reaping the benefits of this very powerful tool set, especially in the final stage of development when real-time analysis is needed most. This application note provides an example of the necessary steps required to utilize DSP/BIOS features in order to bring these solutions to bear in an existing application.

---

## **Contents**

<b>1</b>	<b>Introduction</b> .....	<b>2</b>
<b>2</b>	<b>Zero Degree: 'The Starting Point'</b> .....	<b>2</b>
<b>3</b>	<b>The First Degree: 'Real-Time Printf Debugging'</b> .....	<b>6</b>
<b>4</b>	<b>The Second Degree: 'New Ways to Look at Data'</b> .....	<b>17</b>
	4.1 Analyzing Variables, Execution Time, and Custom Data Displays .....	17
<b>5</b>	<b>The Third Degree: 'Going All the Way'</b> .....	<b>33</b>
	5.1 Using the DSP/BIOS Scheduler .....	33
	<b>Appendix A. Incorporate Pipes into the Audio Application</b> .....	<b>37</b>
	<b>Appendix B. Source Code Listings for Zero Degree – non-BIOS Audio Application</b> .....	<b>43</b>

## **List of Figures**

Figure 1.	Audio Application .....	3
-----------	-------------------------	---

## 1 Introduction

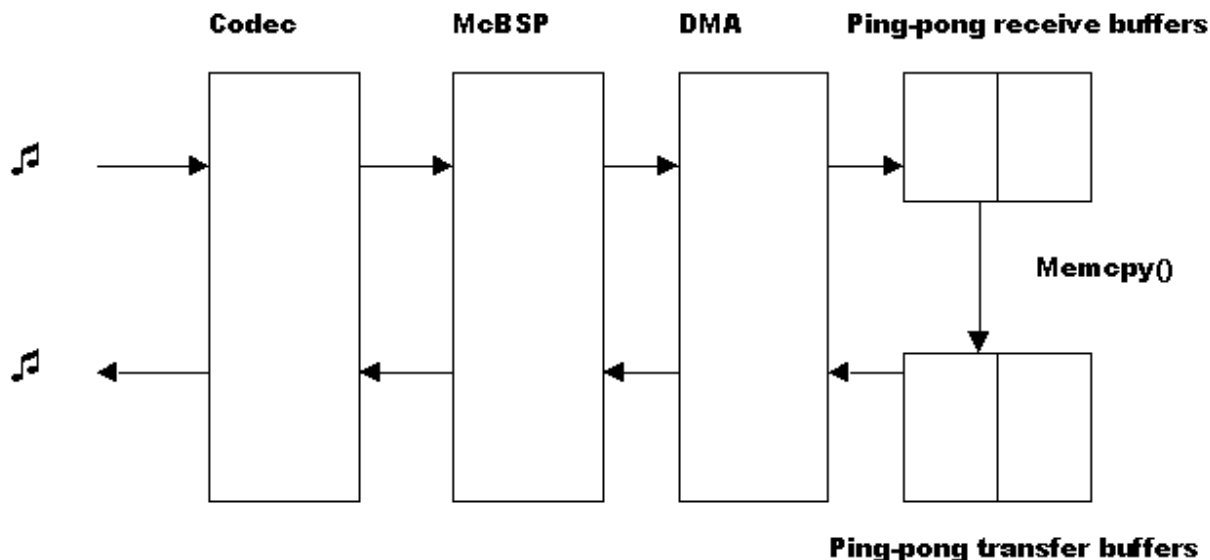
The varying levels of incorporating DSP/BIOS into an application are demonstrated in the following degrees:

- **Zero Degree: ‘The Starting Point’.** This degree introduces a non-BIOS program, ‘audio’, that uses the Multichannel Buffered Serial Port (McBSP) and the Direct Memory Access (DMA) to continuously process digitized audio through the codec on the EVM6x.
- **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 and how to create an RTDX channel to move data between the DSP and a user-created Visual Basic program.
- **Third Degree: ‘Going All the Way’.** The third degree demonstrates how to restructure the application 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 1.2. The audio application is built to run on TI’s EVM6x and uses the codec, Multichannel Buffered Serial Port (McBSP) and direct memory access (DMA) 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.



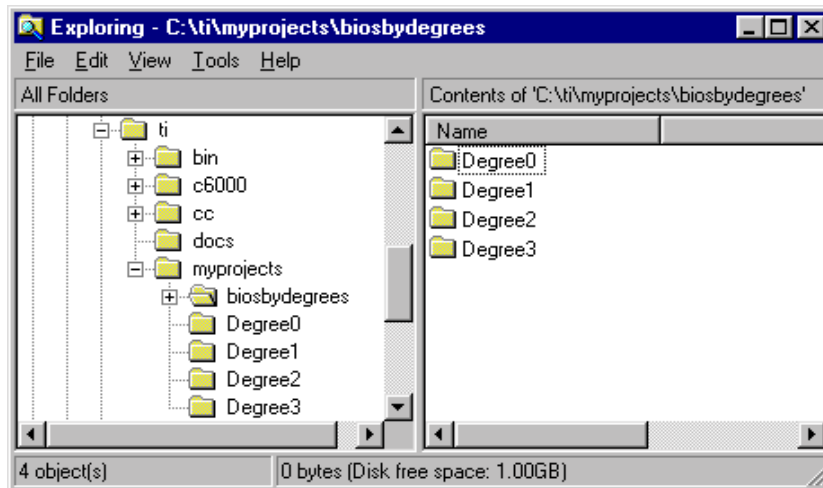
**Figure 1. Audio Application**

Complete source code is provided with this document. Appendix B contains source code for the first task level necessary to incorporate DSP/BIOS into an application; this task level is called Degree0.

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 B to create Degree0 project, or unzip the accompanying zip files, copy only the Degree0 project folder into C:\ti\myprojects\biosbydegrees, and only use the other project folders for reference.


The remainder of this chapter includes instructions on how to create the Degree0 project folder if you choose to cut and copy the code from the Appendix B. 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 B into a project in order to create your starting point. In every degree, you create a new folder and copy 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 three degrees, the directory structure will look like this.



### Step A: Create the necessary source files.

Cut and copy the source code from Appendix B into your own files.

1. Create the directory C:\ti\myprojects\biosbydegrees.
2. In the biosbydegrees folder, make a new folder called Degree0.
3. Browse to Appendix B at the end of this document. Appendix B contains the source code for audio.c, dma.c, init.c, vectors.asm, link.cmd, AudioDMA.h, and dmaistr.s62. 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 change 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 seven files: audio.c, dma.c, init.c, vectors.asm, link.cmd, AudioDMA.h, and dmaistr.s62.

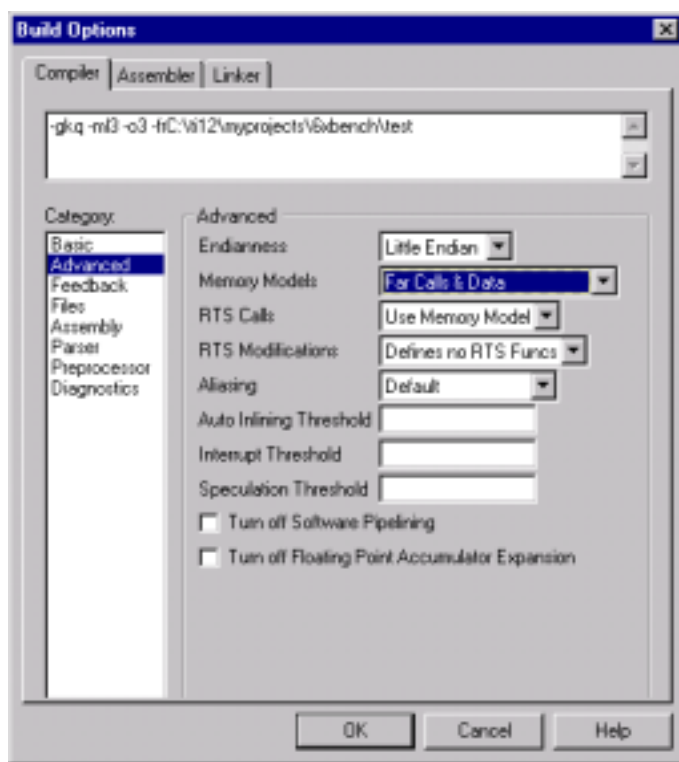
**NOTE:** Pasting from this document into a text editor converts quotes and minus signs into block characters that Code Composer Studio does not recognize. To solve this problem, copy a block character (for example, from the printf() in audio.c) into the Edit → Replace function of your text editor. (Depending 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 Composer Studio.) Replace all block characters with the double quotation mark in audio.c, dma.c, init.c, and vectors.asm. Also, in the enableDMA() function in init.c, search for the assignment statements in the DMA\_GIRA and DMA\_GIRB registers. Both of these statements 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 and dmaistr.s62 because assembly files require that only labels appear in the first column. In vectors.asm and dmaistr.s62, scroll through the text and insert tabs in all lines of text that do not begin with labels. (Reference Appendix B to determine which lines need to be indented.)

### Step B: Create the audio.mak 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 Save In box, select the Degree0 folder. In the File Name box, type audio.mak and click Save.
2. Add the source files to your project. Choose Project → Add Files to Project. Choose audio.c and click Open. Do the same with init.c, dma.c, and vectors.asm.
3. Choose Project → Add Files to Project. Select Linker Command Files (\*.cmd) in the Files of type box. Now choose link.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\cgtools\lib directory. Choose rts6201.lib and click Open. It is not necessary to add AudioDMA.h to the project because it is automatically included during the project build. For now, you do not need to do anything with dmaISR.s62. It is used in Degree Three.
5. Choose Project → Options. Under Category select Advanced. In the Memory Model box, select Far Calls and Data from the pull-down menu. Click OK.



6. Because the program executes out of SDRAM and SBSRAM, it is a good idea to reset the external memory interface (EMIF) when you launch Code Composer Studio and between executions of the program. Click on the + sign next to the Gel files folder in the Project View. Double-click on c6xinit.gel. Look at the StartUp() function and be sure that the emif\_init() is not commented out. If it is, remove the comments. Save and close the file.

The purpose of `emif_init()` in the `StartUp()` function is to initialize the EMIF registers each time Code Composer Studio is launched. Once you have removed the comments, the EMIF should be reset between program executions. You can do so by choosing Gel → Resets → Reset\_and\_EMIF Setup.

7. Reset the EMIF registers now. Choose GEL → Resets → Reset\_and\_EMIF Setup.
8. Choose Project → Rebuild All to build the audio project. Load the program by choosing File → Load Program and by selecting audio out.
9. To hear the audio, you must have a radio or CD player and powered speakers connected to the EVM6x 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.

### 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()` and `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. Use the Help in Code Composer Studio to display additional information on using `LOG_printf()`.

**Table 1. Size / Speed Comparison of LOG\_printf and printf**

	Size (bytes)	Speed (instruction cycles)
<code>Printf()</code>	27K	631
<code>LOG_printf()</code>	132	36

In general, this is the procedure for adding `LOG_printf()` to an existing application:

- 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()`.

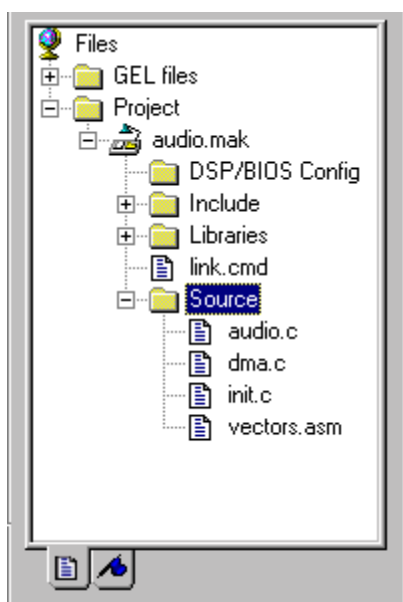
The use of LOG\_printf() requires creating a DSP/BIOS configuration. The configuration is used to create the LOG object as well as to bring in the code for the real-time data link between the host and target. When the configuration is saved, the DSP/BIOS configuration file, \*.cdb, generates three files: \*cfg.s62, \*cfg.h62, 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 required to use the LOG object with LOG\_printf() to replace the original printf() functions.

### Step A: Create a DSP/BIOS configuration file.

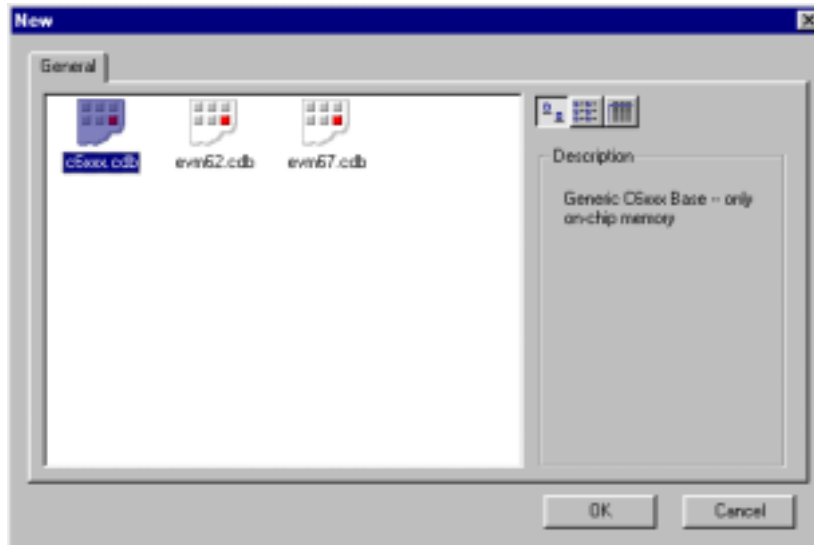
The DSP/BIOS Configuration Tool is first used to configure a LOG object. The unused DSP/BIOS modules do not impact code size; however, the RTDX data pump is automatically included because it transports the LOG\_printf() data from the target to the host.

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.mak 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.mak. This expands the project. Click on the + sign next to the source folder to see all the source files associated with the application.

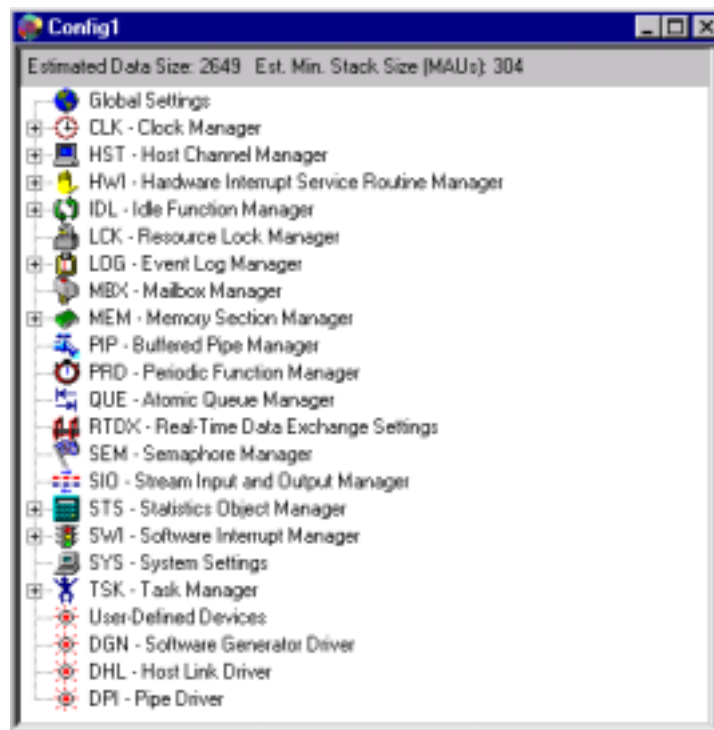


4. Choose File → New → DSP/BIOS Config. Select the generic configuration template entitled C6xxx.cdb and click OK. We are choosing the generic seed file as our starting

point because it is initially configured to use only on-chip memory. This enables you to gain experience using the Configuration Tool to create external memory segments.



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.

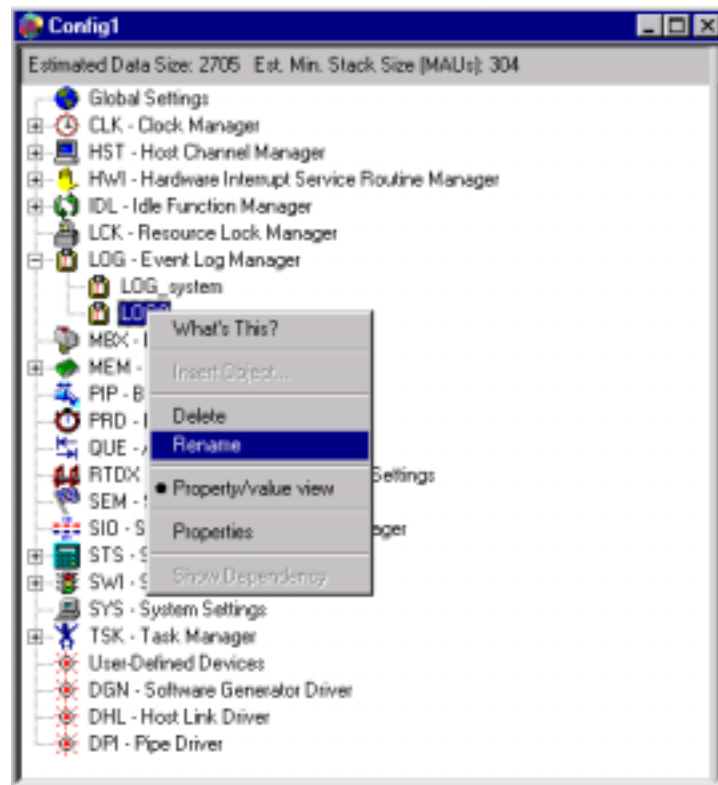




**Step B: Insert a LOG object into the configuration file.**

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

1. 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.



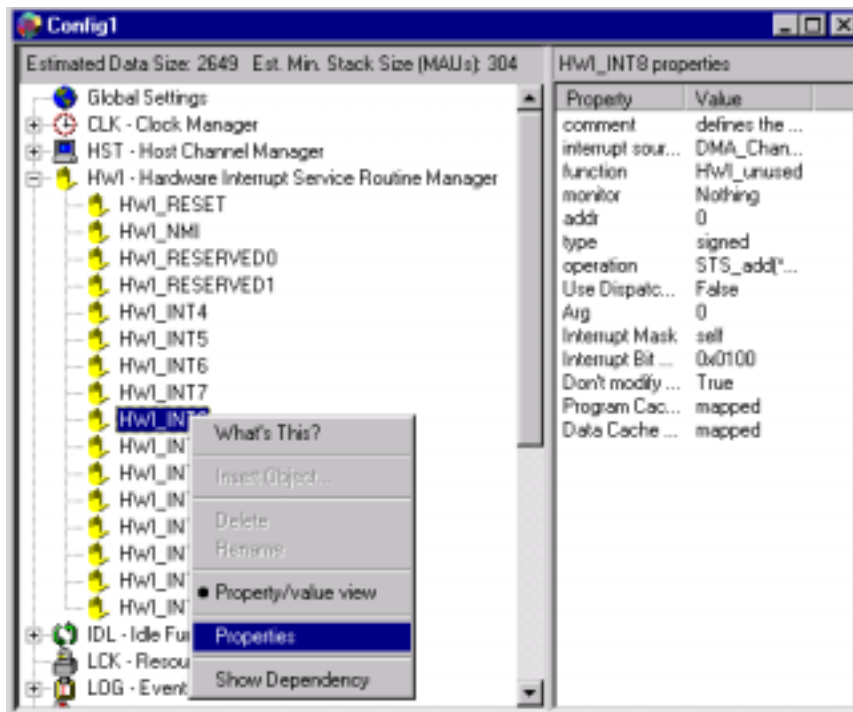
3. Right-click on trace and choose Properties from the pull-down menu. Change the buflen property to 512 words and click OK. This buffer holds (512 words) / (4 words per LOG\_printf) = 128 LOG\_printf messages in a circular fashion.

**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, 9, and reset are handled by interrupt service routines (ISR). The DMA receive ISR, DMArxCisr, uses interrupt 8 and the DMA transmit ISR, DMAtxCisr, uses interrupt 9. Close the file.
2. In the Configuration Tool, 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.



- In the properties window, type the name of the appropriate interrupt service routine, “\_DMARxCisr,” 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, DMA\_Channel\_0, is correct. Click OK.

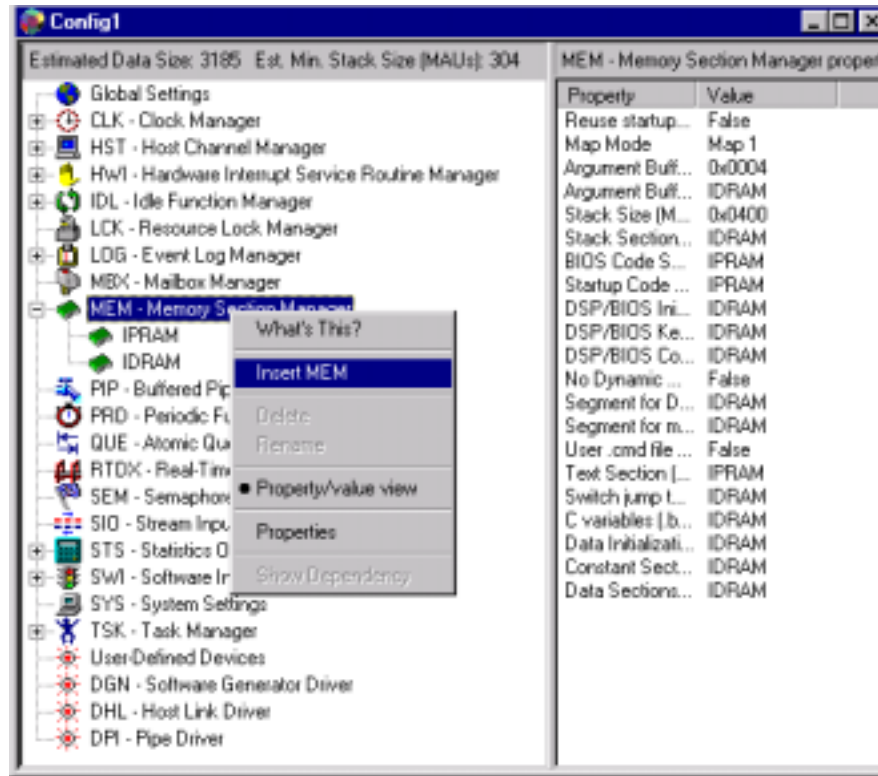


4. Right-click on HWI\_INT9 and choose Properties from the pop-up menu. Type “\_DMAtxCisr” in the function box. Verify that the interrupt source is DMA\_Channel\_1. Click OK. The Hardware Interrupt Service Manager automatically handles the reset interrupt.
5. Click on the – sign next to Hardware Interrupt 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, link.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 Memory Section Manager. The Memory Section Manager has two predefined memory segments, IPRAM and IDRAM. These segments cannot be renamed or removed. They replace the INT\_PROG\_MEM and INT\_DATA\_MEM segments from the original linker command file.
3. Right-click on IPRAM. Choose Properties from the pop-up menu. Since IPRAM corresponds to INT\_PROG\_MEM in link.cmd, confirm that IPRAM has a base address of 0x00000000 and a length of 0x00010000. Click OK.
4. Right-click on IDRAM and choose Properties from the pop-up menu. The IDRAM memory segment corresponds to INT\_DATA\_MEM. Confirm that IDRAM has a base address of 0x80000000 and a length of 0x00010000. Click OK.
5. Since IPRAM and IDRAM replace INT\_PROG\_MEM and INT\_DATA\_MEM in the Configuration Tool, delete the INT\_PROG\_MEM and INT\_DATA\_MEM definitions from the MEMORY directive in link.cmd.
6. Right-click on the Memory Section Manager in the Configuration Tool and select Insert MEM from the pop-up menu.



7. Right-click on the new memory segment and choose Rename from the pop-up menu. Rename the memory object SBSRAM\_PROG\_MEM. (This is the second memory segment defined in link.cmd.) Right-click on SBSRAM\_PROG\_MEM and choose Properties from the pop-up menu.
8. In the properties window uncheck the check box labeled Create a heap in this memory. Set the base equal to 0x00400000 and the length to 0x00014000. Select code to designate the type of memory space you are defining. Click OK. Delete the SBSRAM\_PROG\_MEM definition from the linker command file.



9. Continue to transfer the following memory regions into the Configuration Tool as you did in steps 6, 7, and 8: SBSRAM\_DATA\_MEM, SDRAM0\_DATA\_MEM, and SDRAM1\_DATA\_MEM. For these three segments, leave the space property defined as data. Create the memory segment SBSRAM\_DATA\_MEM and set its base address to 0x00414000 and its length to 0x0002C000. Create the memory segment SDRAM0\_DATA\_MEM and set its base address to 0x02000000 and its length to 0x00400000. Create the memory segment SDRAM1\_DATA\_MEM and set its base address to 0x03000000 and its length to 0x00400000. Delete each 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.
10. Refer to the linker command file, link.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 drop-down list next to the 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.)
11. Assign the Text Section (.text), the BIOS Code Section (.bios) and the Startup Code Section (.sysinit) to SBSRAM\_PROG\_MEM.
12. Assign the Data Section (.data, .switch, .cio, .systemem) to SDRAM0\_DATA\_MEM. The default settings for .const, .bss, .cinit, .pinit, .stack, and .far correspond to the settings of the original linker command file. In addition, the interrupt vector table is automatically loaded at address 0, so no further changes to the section settings are necessary.
13. Except for the sbsbuf section, remove all section assignments from within the SECTION directive of link.cmd. They are no longer necessary because the Configuration Tool now properly assigns these sections. The philosophy is to let the Configuration Tool show all the common sections and to leave any custom definitions in the application linker command file.

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

Saving the configuration generates the three files: audiocfg.s62, audiocfg.h62, 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.

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

### 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.

- Add "-laudiocfg.cmd" 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.
- Save the file and exit the text editor. The link.cmd file should now look like:

```
/* linker command file for audio */
-laudiocfg.cmd
SECTIONS
{
    sbsbuf          load = SBSRAM_DATA_MEM
                   { _SbsramDataAddr = .; _SbsramDataSize = 0x0002C000; }
}
```

### 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(dataReadyFlag)
    {
        dataReadyFlag = 0;
        processBuffer();
    }
    IDL_run();
}
```

4. Remove the call to `initInterrupts()` from `audio.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`, the DMA setup continues to enable hardware interrupts 8 and 9 because the DMA channels use them.

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

Use `LOG_printf()` to replace the `printf()` functions 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 require first including `std.h`. Add the 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 <string.h>
#include <math.h>
#include "AudioDMA.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 calls to LOG\_printf() as shown in the example below. Remember to delete the original printf() function calls. Save audio.c when you are done.

```
void main()
{
    int date = 25;
    float percent = 10.26;
    int time;
    BIOS_start();
    /* This requires two LOG_printf() functions because
       LOG_Printf() has a limit of two parameters. Refer to
       help or the DSP/BIOS User's Guide for details */
    LOG_printf(&trace, "Why do computer scientists celebrate Halloween and Christmas on
the same day?\
\nBecause oct %o equals dec %d.\n", date, date);

    LOG_printf(&trace, "(%6.2f percent of the population gets this joke.)\n", percent);

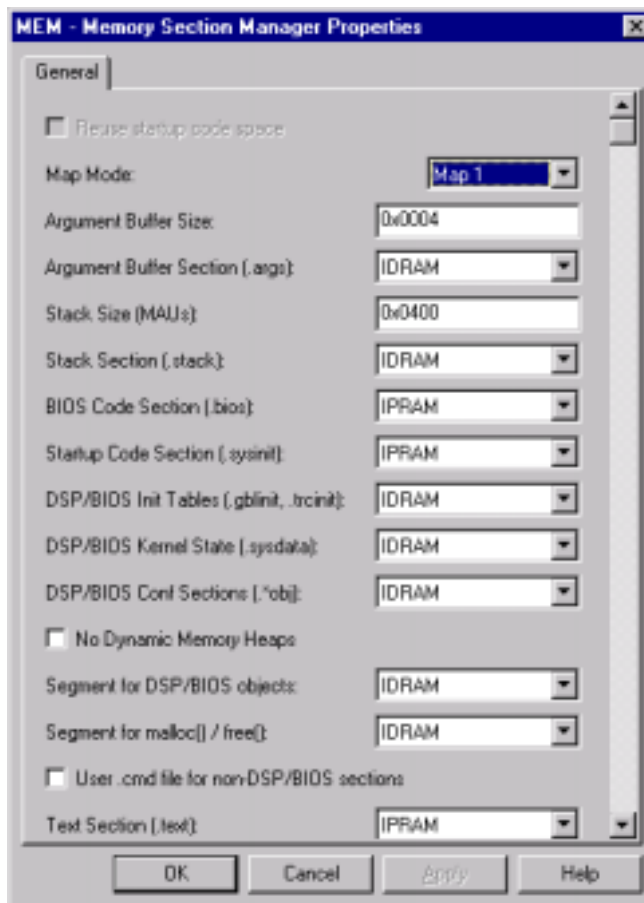
    initApplication();

    /* Start DMA with autoinitialization */
    DMA_rxStart((unsigned int)&ping_RX[0], DMA_FRAME_SIZE);
    DMA_txStart((unsigned int)&ping_TX[0], DMA_FRAME_SIZE);
    LOG_printf(&trace, "DMA started.\n");
}
```

### Step I: Run the program.

1. Choose Project → Rebuild All.
2. Choose File → Load Program and select audio.out. Click Open.
3. Choose Tools → DSP/BIOS → Message Log. Right-click on the Message Log window and select Property Page from the pop-up menu. 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.
4. Choose Debug → Go Main, and then Debug → Run.
5. Finally, because the program is small enough to run out of internal memory, move the program back into internal memory space. Now that you have the experience of designating memory segments and sections, you can use that ability with larger programs that require additional space. Choose Debug → Halt. Double-click on audio.cdb. Right-click on the Memory Section Manager. Choose Properties from the pop-up menu. Change all sections so that the properties correspond to the example below. When you are finished, click OK. Choose File → Save to save the changes to the configuration.





## 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 two examples also show how to perform hardware interrupt logging and how to use the RTDX channels to move data between the DSP and a Visual Basic program.

### 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.mak 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.mak. 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. Notice the sine wave generator function, generateSine(), 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 first two LOG\_printf() functions.
5. Double-click on audio.cdb to open the Configuration Tool.
6. Right-click on the Statistics Object Manager and choose Insert STS from the pop-up menu.
7. 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.
8. Choose File → Save to save the changes to the configuration. Close the tool.
9. Add the necessary DSP/BIOS header file to audio.c. Remember that this addition must be included after std.h.

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

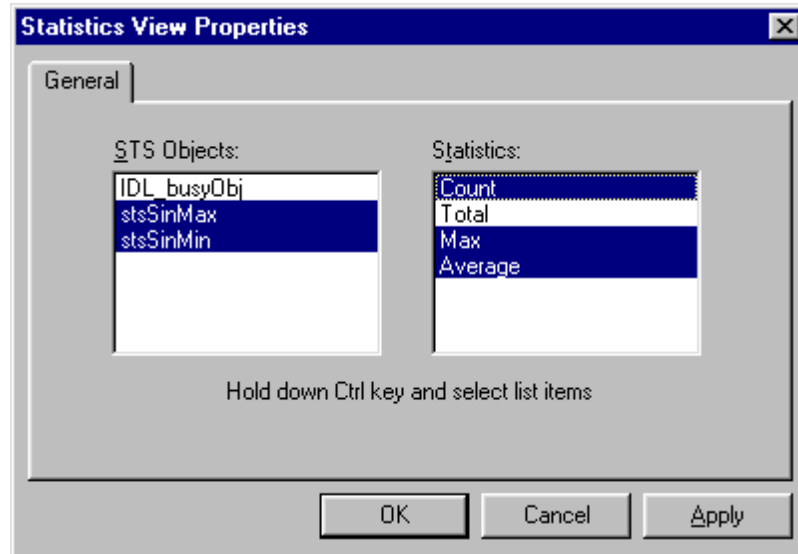
10. Declare the statistics objects in audio.c by adding the following lines above the main() function. Now that the entire program is stored in internal memory, the far keyword is no longer necessary; however, it is still used so that if the code is moved later, it can be done so without encountering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

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

11. 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;
}
```

12. Choose File → Save.
13. Choose Project → Rebuild All and load the executable audio.out.
14. Choose Tools → 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 and select the count, maximum, and average statistics. Click OK.



15. Choose Debug → Go Main and then Debug > Run.
16. 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 sineValue 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.
17. Choose Debug → Halt.
18. 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;
    float percent = 10.26;
    int time;

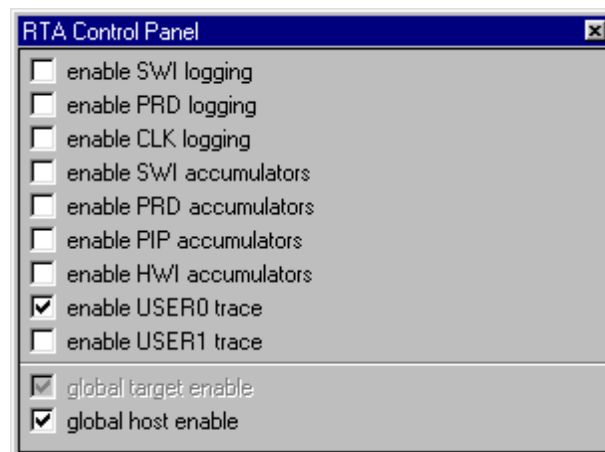
    BIOS_start();

    if (TRC_query(TRC_USER0) == 0)
    {
        time = CLK_geththtime();
        STS_set(&stsLogPrintf, time);
    }
    LOG_printf(&trace, "Why do computer scientists celebrate Halloween and Christmas on
the same day?\n
    \nBecause oct %o equals dec %d.\n", date, date);

    LOG_printf(&trace, "(%.2f percent of the population gets this joke.)\n", percent);
    if (TRC_query(TRC_USER0) == 0)
    {
        time = CLK_geththtime();
        STS_delta(&stsLogPrintf, time);
    }
}

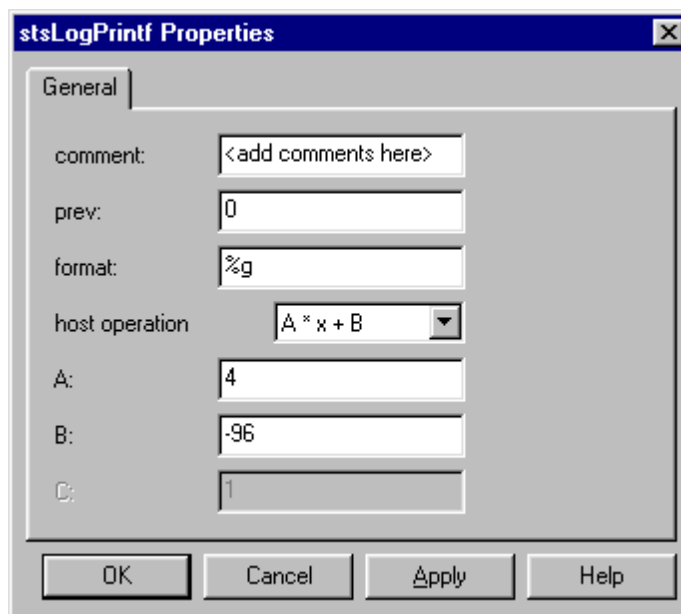
```

6. Comment out the lines of code for the two calls 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 Project → Rebuild All. Load the program.
9. Choose Tools → 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.
10. 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\_geththtime(), 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*, literature number SPRU328.

11. Choose Tools → DSP/BIOS → Statistics View. Right-click on the Statistics View area and choose Property Page from the pop-up menu. Select stsLogPrintf and select Count, Max, and Average from the available statistics. Click OK.
12. Choose Debug → Go Main and then Debug → Run.
13. The Statistics View reports an Average and Max value of 24. 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 24 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 96 execution cycles. Choose Debug → Halt.
14. 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 -96. Click OK.

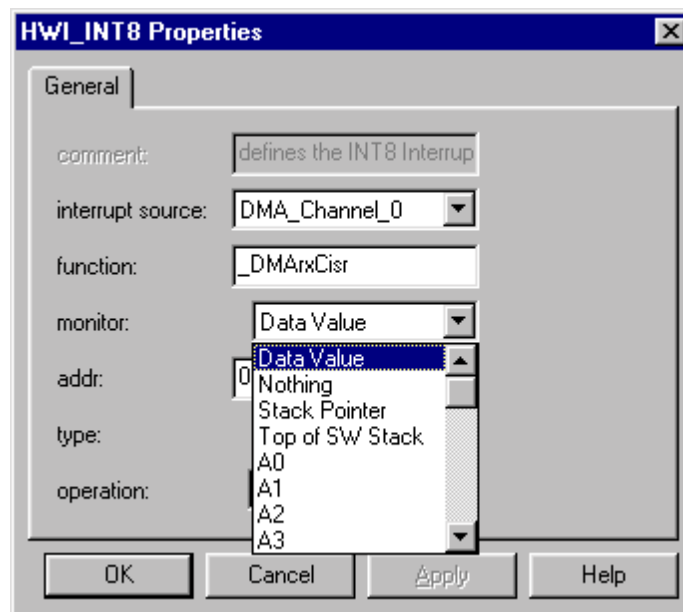


15. Save the configuration file.
16. In audio.c, uncomment the calls to LOG printf(). Choose Project → Rebuild All. Reload the executable and run. Notice that the two calls to LOG printf() require only 72 instruction cycles.
17. Choose Debug → Halt. Right-click on the Statistics View area and choose Close. Right-click on the Control Panel and choose Close.

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

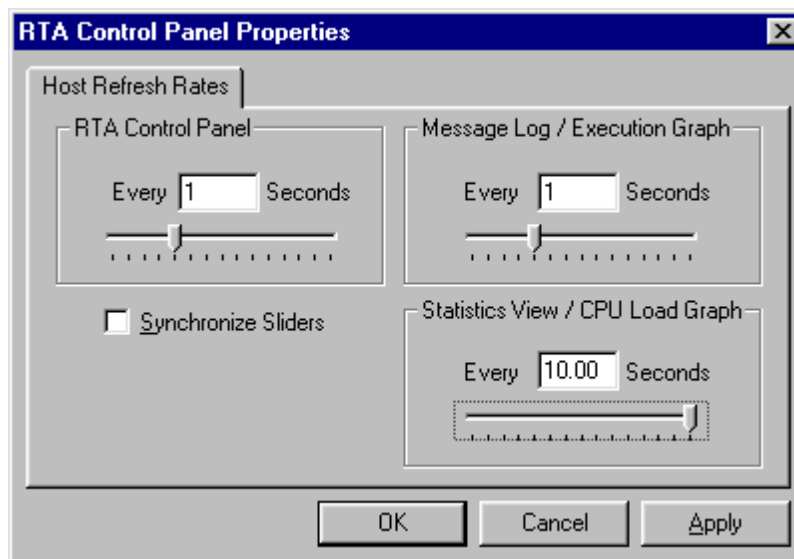
HWI monitoring is enabled through the DSP/BIOS Configuration Tool and no API needs to be added. This is known as implicit instrumentation as opposed to explicit which was described previously in Step B.

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. Right-click on HWI\_INT9. Select Properties from the pop-up menu. In the monitor list box menu, select Data Value and click OK.
3. Notice that this results in the creation of two new statistics objects, HWI\_INT8\_STS and HWI\_INT9\_STS. Choose File → Save and close the Configuration Tool.
4. Choose Project → Rebuild All. Reload the executable.
5. Choose Tools → DSP/BIOS → RTA Control Panel. Put check marks in the boxes next to global host enable and enable HWI accumulators.
6. Choose Tools → 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 and HWI\_INT9\_STS objects and select the Count statistic.
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.

#### Step D: Custom Data Displays and Controls using RTDX and Visual Basic.

Gathering data for analysis and setting DSP parameters in real time can be done using RTDX from Visual Basic. RTDX gives the user the ability to establish 'virtual serial ports' for moving data to and from the target application and the host PC. The Learning Edition of Visual Basic 6.0 is the minimum requirement for this step in which you create a simple program that controls the volume of the audio application.

1. Create RTDX channels in the audio application using the RTDX Macros. Double-click on audio.c in the Project View. Copy the code below and paste it into audio.c above the main() function.

```
/* RTDX channels structures (these are macro calls) */
RTDX_CreateInputChannel(control_channel);
RTDX_CreateOutputChannel(status_channel);
```

2. Add the RTDX header.

```
#include <rtdx.h>
```

3. Declare the global variables shown below in audio.c.

```
int volume = 1;
int average = 0;
int count = 0;
int accum = 0;
```

4. Open the RTDX channels in main(). Add the code before the call to BIOS\_start().

```
// enable the RTDX control channels
RTDX_enableInput(&control_channel);
RTDX_enableOutput(&status_channel);
```

5. Add an RTDX command handler. RTDX commands are issued to the target one integer (four bytes) at a time. For simplicity, the opcode is packed into the high order byte, and the data is the low order three bytes. The RTDX command handler reads the RTDX command from the channel and separates the command from the data. Based on the command received, it takes appropriate action. In this example, the command opcode changes the volume and the data contains the volume level, but this approach can also be used to apply filters. Copy and paste the code below into audio.c. Remember to replace the quote characters in the LOG\_printf() functions.

```
/*
 * ===== RTDXcommandHandler =====
 *
 * This function is called from the IDL Module to handle RTDX
 * commands sent from the Host
 *
 */
void RTDXcommandHandler()
{
    static unsigned int control, opCode;
    static unsigned int rxInt = 0;
    /* Read new load control when host sends it */
    if (!RTDX_channelBusy(&control_channel))
    {
        // read integer from control channel
        RTDX_readNB(&control_channel, &rxInt, sizeof(rxInt));
        // opcode and data (control) are packed together into one
        // 32bit word
        opCode = rxInt>>24;
        control = (int)(rxInt & 0x00ffffff);
        switch(opCode)
        {
            case VOLUME__OPCODE: // 0x70
                volume = control;
                LOG_printf(&trace, "Recv known RTDX: OpCode = 0x%02x, Control = %d",
opCode, control);
                break;

            default:
                LOG_printf(&trace, "Recv Unknown RTDX: OpCode = 0x%02x, Control = %d",
opCode, control);
                break;
        }
    }
}
```



6. Change the processBuffer() function to respond to the changes in the volume and to send information about the average sample back to the Visual Basic program. ProcessBuffer() must multiply the sampled sound data by the volume variable before copying the data into the transmit buffer. In addition, processBuffer() also accumulates the average of the absolute value of the audio samples and sends that value back to the Visual Basic program where it is displayed. Cut and paste the code below to replace the existing processBuffer() function.

```

/* ===== processBuffer ===== *
 * FUNCTION: copy RX buffer to TX buffer
 * PARAMETERS: none
 * RETURN VALUE: none
 */
void processBuffer(void)
{
    extern int ping_tx_flag;
    extern int ping_rx_flag;
    int index;
    int total = 0;
    short *src, *dst;

    if(ping_rx_flag == 1)
        src = (short *)&pong_RX[0];
    else
        src = (short *)&ping_RX[0];

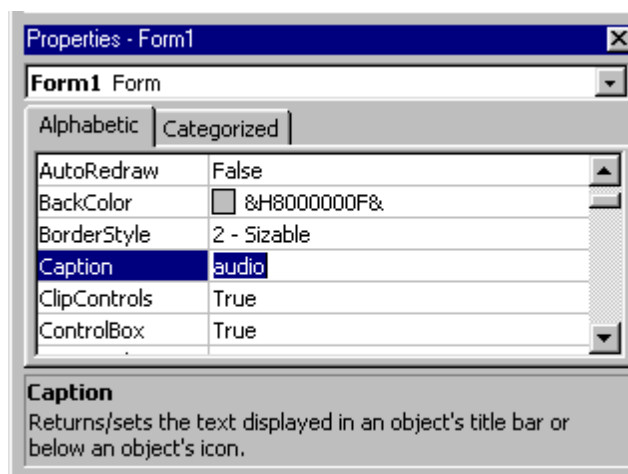
    if(ping_tx_flag == 1)
        dst = (short *)&pong_TX[0];
    else
        dst = (short *)&ping_TX[0];

    for(index = 0; index < DMA_FRAME_SIZE * 2; index++)
    {
        *dst++ = (*src++ * volume);
        if(*dst < 0)
        {
            total += ~(*dst + 1);
        }
        else
        {
            total += (*dst);
        }
    }
    count++;
    accum = accum + (total / (DMA_FRAME_SIZE * 2));
    if(count == 10)
    {
        average = accum/count;
        RTDX_write(&status_channel, &average, sizeof(int));
        count = 0;
        accum = 0;
    }
}

```

7. Click on the + sign next to the Include folder in the Project View. Double-click on AudioDMA.h to open the file. Add the definition of the volume opcode to the file.
 

```
#define VOLUME__OPCODE    0x70
```
8. Choose File → Save and close AudioDMA.h. Click on the – sign next to the Include folder to collapse the list of header files.
9. Add an IDL object to the Configuration Tool that calls RTDXcommandHandler(). The IDL module manages low priority threads that only run when no hardware, software interrupts, or tasks need to run. By making RTDXcommandHandler() an idle thread, you ensure that the RTDX command handler does not interfere with the real-time deadlines of the application. Double-click on audio.cdb. When the Configuration Tool opens, right-click on the Idle Function Manager. Choose Insert IDL from the pop-up menu.
10. Right-click on the new IDL object, IDL0, and choose Rename from the pop-up menu. Rename the object idlRTDXcommandHandler.
11. Right-click on idlRTDXcommandHandler and choose Properties from the pop-up menu. In the function box, type “\_RTDXcommandHandler.” Click OK.
12. Choose File → Save and close the Configuration Tool.
13. Now you need to create the Visual Basic program. This program manages the host side of the RTDX channels. Begin by launching Visual Basic 6.0. (You do not need to close Code Composer Studio.) In the Visual Basic tool, choose File→New Project and select ‘Standard EXE.’ Click OK.
14. An empty form and project appears in the workspace. Choose File – > Save Project and save the empty form as audio.frm and the project as audio.vbp into the C:\ti\myprojects\biosbydegrees\degree2 directory.
15. Notice the Properties window on the right side of your screen. The Properties window currently displays the properties for Form1 (audio.frm). It provides you with direct access to all the properties so that you can easily examine or change them. Find the property called Caption. Next to Caption, replace ‘Form1’ with ‘audio.’ Notice that this changes the caption on your form from Form1 to audio.



16. Double-click on the empty form, and a code window pops up. Select all source code that is currently in the code window. Then copy the code that follows this step and paste it over the original code in order to replace it. Notice the creation of the RTDX channels in the

Form\_Load subroutine. Form load is the code that is executed when the application starts and the form is loaded.

**NOTE:** Pasting from this document into Visual Basic causes problems with the single and double quote characters. Replace the single quote character with the single quote character on the keyboard (the key directly to the left of the Enter key). Replace the double quote characters as well. In addition, some lines of source code wrap onto the next line in this document, which causes compile errors in Visual Basic. These lines continue to appear in red after you have replaced all quote characters. Find these lines and remove the carriage return so that the source code does not wrap onto the next line in the Visual Basic code window.

```
' RTDX OLE API Status Return codes
Const SUCCESS = &H0 'Method call successful
Const FAIL = &H80004005 'Method call failure
Const ENoDataAvailable = &H8003001E 'No data is currently available
Const EEndOfLogFile = &H80030002 'End of log file
' Upper byte of RTDX control word is the opcode
Const VolumeOpCode = &H70000000
' Channel name constants
Dim bufstate As Long
' RTDX COM Objects, one per channel.
Dim rtdxToDSP As Object
Dim rtdxFromDSP As Object
Private Sub Form_Load()
    Dim status As Long
    Dim response As Integer ' Response variable for message box

    ' Create instances of the RTDX COM object,one for read channel, one for write
    Set rtdxToDSP = CreateObject("RTDX") ' Create RTDX write channel
    On Error GoTo On_Error ' Test object instantiation

    Set rtdxFromDSP = CreateObject("RTDX") ' Create RTDX read channel
    On Error GoTo On_Error ' Test object instantiation

    ' Open the RTDX read channel used to get status from the DSP app
    status = rtdxFromDSP.Open("status_channel", "R")
    Select Case status
        Case Is = SUCCESS
        Case Is = FAIL
            response = MsgBox("Error: Opening status_channel failed", vbCritical)
            Exit Sub
        Case Else
            response = MsgBox("Error: Unknown value (" + status + ") from status_channel
open", vbInformation)
            Exit Sub
    End Select

    ' open the RTDX write channel used to control the DSP application
    status = rtdxToDSP.Open("control_channel", "W")
    Select Case status
        Case Is = SUCCESS
        Case Is = FAIL
            response = MsgBox("Error: Opening control_channel failed", vbCritical)
            Exit Sub
        Case Else
```

```

        response = MsgBox("Error: Unknown value (" + status + ") from control_channel
open" + " open", vbInformation)
        Exit Sub
    End Select
    GoTo EndLabel

On_Error:
    response = MsgBox("Error: Instantiation Failed", vbCritical)
EndLabel:

End Sub
Private Sub Form_Unload(Cancel As Integer)
    ' close target's input channel

    status = rtdxToDSP.Close()
    Select Case status
        Case Is = SUCCESS
        Case Is = FAIL
            MsgBox "Unable to close channel to DSP", vbCritical, "Error"
        Case Else
            MsgBox "Unknown return closing channel to DSP", vbInformation
    End Select

    Set rtdxToDSP = Nothing                ' kill RTDX OLE object
    status = rtdxFromDSP.Close()
    Select Case status
        Case Is = SUCCESS
        Case Is = FAIL
            MsgBox "Unable to close channel from DSP", vbCritical, "Error"
        Case Else
            MsgBox "Unknown return value closing channel from DSP", vbInformation
    End Select

    Set rtdxFromDSP = Nothing            ' kill RTDX OLE object
End Sub
Private Sub OnTop_Click()
    Dim lR As Long

    ' Always On Top Check Box
    If OnTop.Value = 0 Then
        lR = SetTopMostWindow(Form1.hwnd, False)
    Else
        lR = SetTopMostWindow(Form1.hwnd, True)
    End If
End Sub
Private Sub Slider1_Click()
    Dim status As Long
    Dim dataI4 As Long
    ' write volume selection to target input channel
    dataI4 = VolumeOpCode + Slider1.Value

    'dataI4 = Slider1.Value
    status = rtdxToDSP.WriteI4(dataI4, bufstate)

    Select Case status
        Case Is = SUCCESS
        Case Is = FAIL
            MsgBox "Error: Writing data failed" & dataI4

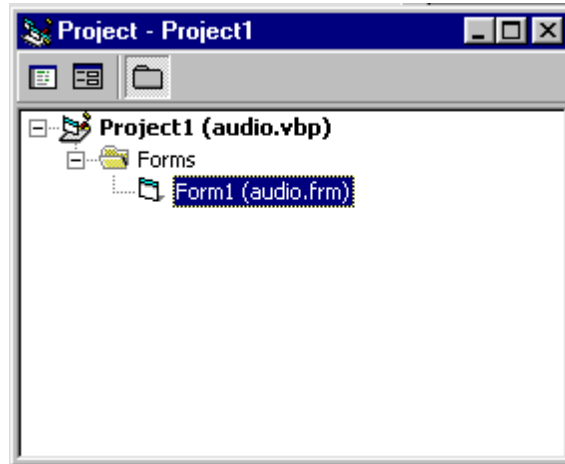
```

```

Case Else
    MsgBox "Target control data backed up waiting for DSP to read it"
End Select
End Sub

```

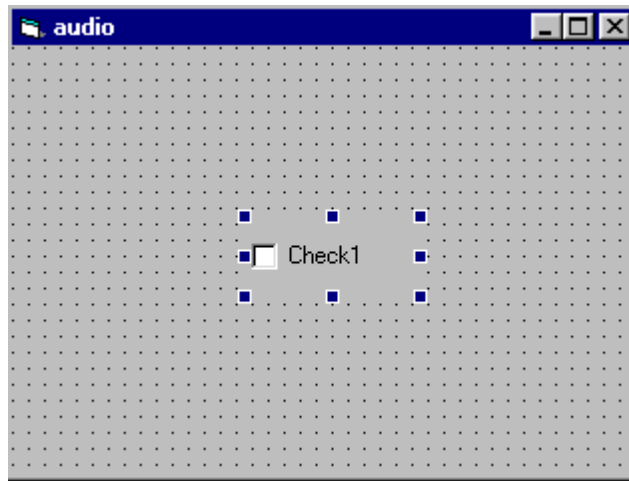
17. In the Visual Basic Project window, double-click on Form1 (audio.frm) to bring up the form in the workspace. The Visual Basic Project window is shown in the picture below.



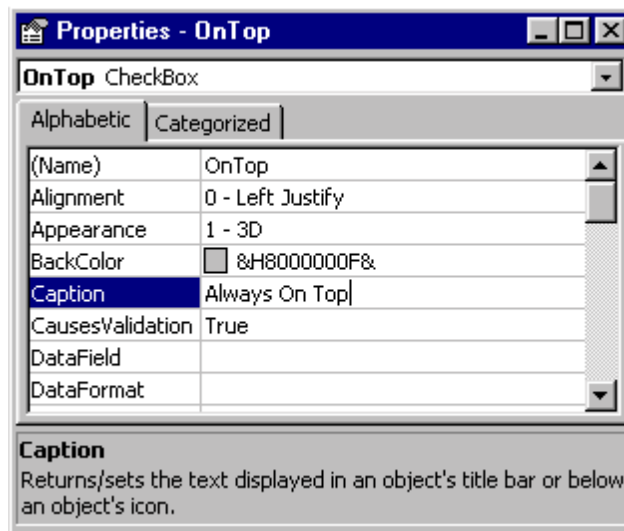
18. On the left side of the workspace is the toolbox. This displays the various components currently available to add to your form. (If the toolbox is not visible, choose View → Toolbox.) Your toolbox may have different components than the toolbox shown here.



19. You can learn the name of an item on the toolbox by placing the mouse arrow over the component. The name appears in a tool tip. Find the CheckBox component. Double-click on CheckBox. This causes a check box to appear on the form.



20. Because the check box is selected, its properties appear in the Properties window. In the Properties window, change the Name property to OnTop. Change the Caption property to Always On Top. You may need to resize the check box so that it can display the caption correctly.



21. Insert a Module into the project so that when the check box is checked, it enables your Visual Basic program to remain on top of all other windows. This is convenient when you run the program and Code Composer Studio together. Choose Project → Add Module. Select the Module and click Open. (Make sure the New tab is selected, not Existing.) This results in an open code window for the new module. Copy and paste the code below into the module's code window. After pasting this code into the Visual Basic project, you must replace single quote, double quote, and minus signs in order to avoid compile errors.

```

Option Explicit
Public Const SWP_NOMOVE = 2
Public Const SWP_NOSIZE = 1
Public Const FLAGS = SWP_NOMOVE
Public Const HWND_TOPMOST = -1
Public Const HWND_NOTOPMOST = -2
Declare Function SetWindowPos Lib "user32" (ByVal hwnd As Long, _
    ByVal HwndInsertAfter As Long, _
    ByVal x As Long, _
    ByVal y As Long, _
    ByVal cx As Long, _
    ByVal cy As Long, _
    ByVal wFlags As Long) As Long

    'SetTopMostWindow = SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 470, 315, FLAGS)

Public Function SetTopMostWindow(hwnd As Long, Topmost As Boolean) As Long
    If Topmost = True Then 'Make the window topmost
        SetTopMostWindow = SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 700, 240, FLAGS)
    Else
        SetTopMostWindow = SetWindowPos(hwnd, HWND_NOTOPMOST, 0, 0, 700, 240, FLAGS)
        SetTopMostWindow = False
    End If
End Function

```

22. Choose File → Save Module1. Save the module in your working directory, Degree2, with the default name, Module1.bas.
23. Double-click again on Form1 (audio.frm) in the Project window. Examine the items in your toolbox by placing the mouse cursor over the item until the name of the item appears. Look for a component called 'Slider.' If the slider tool is not currently available in your toolbox, choose Project → Components. Select Microsoft Windows Common Controls 5.0 (SP2). (Make sure that the Controls tab is selected.) Click OK. This adds additional components to your toolbox.
24. Double-click on the slider in the toolbox in order to add it to the form. The slider may appear on the form placed directly on top of the check box. If so, you can move the slider by clicking and dragging.
25. In the Properties window, change the slider's Orientation property from sldHorizontal to sldVertical.
26. Find the component in the toolbox called Timer. Double-click on the timer to make a timer appear on the form. In the Properties window, change the timer's Interval property to 500. This is the number of milliseconds between timer events.
27. Find the component in the toolbox called ProgressBar. Double-click on the ProgressBar to make one appear on the form. In the Properties window, change the Width property to 7900, and the Height to 400. Change the Max property to 2300. Enlarge the form so that you can see the entire progress bar by clicking and dragging the edges of the form to expand the area.
28. Double-click on the timer on the audio form. This takes you to the code window and displays the function that runs when the timer event occurs. Right now the function is empty. Cut and copy the code below into the space between Private Sub Timer1\_Timer() and End Sub. Remember to change the single and double quote characters in order to avoid compile errors.

```

' read RTDX channel for status information
' update the progress bar during recording

#####
' This procedure is enabled(called repeatedly) when the user starts the
' the test. It acts as a task and is not halted until the user disables
' the timer (i.e. TimerRTDX.Enable = "False"). This
' procedure's job is grab data from the running target for status
#####
  Dim dataI4 As Long
  Dim status As Long

  status = rtdxFromDSP.ReadI4(dataI4)

Do Until status = ENoDataAvailable

  Select Case status
    Case Is = SUCCESS
      If dataI4 > 2300 Then dataI4 = 2300
      If dataI4 < 0 Then dataI4 = 0
      ProgressBar1.Value = dataI4

    '-----
    Case Is = FAIL
      MsgBox "Error: Reading data failed" & dataI4
      TimerRTDX.Enabled = False ' Disable timer method dispatch
    Case Is = ENoDataAvailable
      ' No Data is available

    Case Else
      MsgBox "Target control data backed up waiting for DSP to read it" +
Hex(status)
      TimerRTDX.Enabled = False ' Disable timer method dispatch
  End Select
  status = rtdxFromDSP.ReadI4(dataI4)
Loop

```

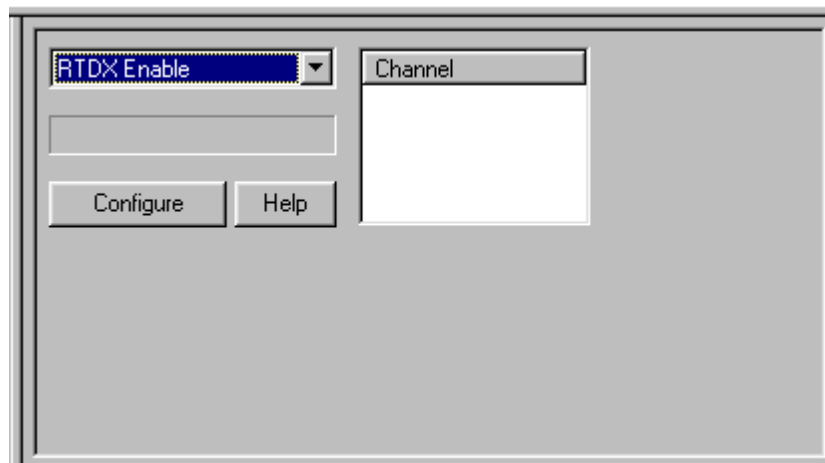
29. Choose File → Save Project.

30. Choose File → Make audio.exe and save the executable into Degree2. Exit Visual Basic. If you are prompted to save changes to audio.frm and audio.vbp, choose Yes.

31. Return to the Code Composer Studio window. Choose Project → Rebuild All. Load the executable, audio.out. It is important to load the Code Composer Studio executable before launching the Visual Basic audio.exe. This is because when the Visual Basic form loads, it immediately attempts to connect to the RTDX channels. This results in an error if the audio application is not yet loaded because it creates the channels.

32. Choose Tools → RTDX. Verify that RTDX is enabled. If it is not, select RTDX Enable from the pull-down list. Right-click on the RTDX window and select Hide.





33. Choose Tools → DSP/BIOS → Message Log. Right-click on the Message Log window and select Property Page from the pop-up menus. On the Properties page, select trace as the name of the Log Object and click OK.
34. Choose Debug → Go Main. Then choose Debug → Run.
35. Using Windows Explorer, browse to the Degree2 folder and double-click on the Visual Basic program, audio.exe. The audio form appears.
36. Use the slider to adjust the volume of the audio application. Realize that the Visual Basic control does not send any information to the audio application until you activate the control by adjusting the slider.
37. Choose Debug → Halt. Close audio.exe.

## 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, multichannel applications, etc.)

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. SWIs are a good choice when space is limited. SWIs are modeled after hardware interrupt service routines. SWIs run to completion, share the system stack with hardware interrupts (HWIs), and are prioritized. All SWIs run at a higher priority than tasks. Tasks are more akin to traditional main loop programming, because they run a processing loop. Tasks offer more flexibility than SWIs; they have synchronization and communication mechanisms, they each have their own stack, and they can block. As a result of the increased flexibility, tasks are also more demanding on system resources than SWIs.

It is the scheduler's job to manage the execution of the DSP/BIOS threads. When the scheduler runs, it checks the priorities of readied threads to determine which to run next, and whether to preempt the currently running thread. When no threads are pending, the idle loop runs through the idle functions configured under the IDL Manager.

In this example, SWIs are created to perform the work previously done in main().

### Step A – Remove the processing from main():

The first step is to remove the majority of the processing from main() so that it 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.mak file to open the project.
3. In the Project View, click on the + sign next to the Project folder and then next to audio.mak to expand the project. Click on the + sign next to the Source folder and the DSP/BIOS Config folder.
4. Double-click on audio.cdb. In the Configuration Tool, click on the + sign next to the IDL Function Manager to expand the list of IDL objects.
5. Right-click on idIRTDXcommandHandler and choose Delete from the pop-up menu. Now that Degree 2 has been completed, this IDL object is no longer necessary. Save and close the configuration file.
6. Double-click on audio.c to open the source file.
7. 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;
    float percent = 10.26;

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

    LOG_printf(&trace, "(%6.2f percent of the population gets this joke.)\n",
    percent);

    /* Configure Peripherals */
    initApplication();

    /* Start DMA with autoinitialization */
    DMA_rxStart((unsigned int)&ping_RX[0], DMA_FRAME_SIZE);
    DMA_txStart((unsigned int)&ping_TRX[0], DMA_FRAME_SIZE);

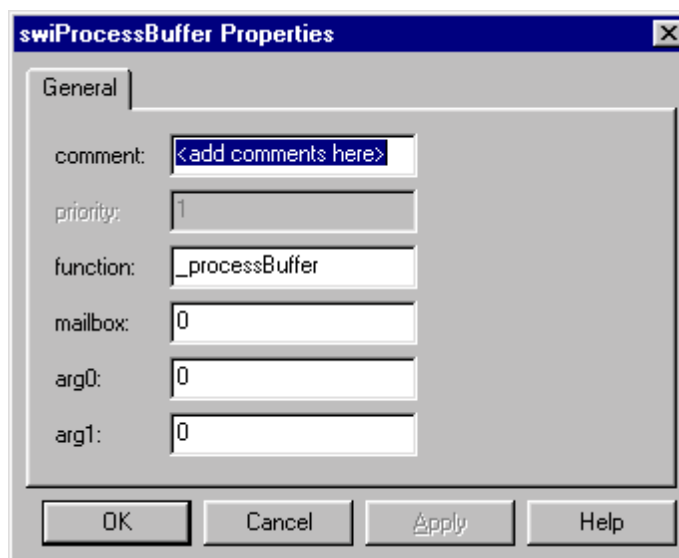
    /* Fall into DSP/BIOS idle loop */
    return;
}

```

### 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 DMA 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. Click OK.



### Step C: Modify DMARxCisr() to post the software interrupt.

Now that processBuffer() runs as result of a software interrupt, the interrupt service routine for the receiving DMA channel must post the software interrupt.

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

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

3. Modify DMARxCisr() in dma.c by calling SWI\_post() and removing the assignment into dataReadyflag as in the example below.

```

interrupt void DMARxCisr(void)
{
    /* enable end-of-frame interrupt */
    DMA0_SCNTL = SCNTL_ENABLE;
    /* post software interrupt swiProcessBuffer */
    SWI_post(&swiProcessBuffer);
    /* ping_rx_flag starts at 1 and toggles with each interrupt */
    if(ping_rx_flag == 1)
        ping_rx_flag = 0;
    else
        ping_rx_flag = 1;
}

```

4. In the next step you create assembly ISRs that branch to DMARxCisr and DMAtxCisr. To branch to DMARxCisr and DMAtxCisr from the assembly, you must first remove the word 'interrupt' from their definitions. Remove the word 'interrupt' from the DMARxCisr and DMAtxCisr definitions in the dma.c source file.
5. Choose File → Save. Close the file.

#### **Step D: Create the assembly interrupt service routines to branch to the original ISRs.**

Now that the ISR for DMA Channel 0 posts a software interrupt, it is necessary either to use the assembly macros, HWI\_enter and HWI\_exit, or to use the HWI dispatcher. This example uses the assembly macros instead of the dispatcher. The purpose of both the dispatcher and the macros, HWI\_enter and HWI\_exit, is to store and restore the process context when using ISRs, and to make sure that the scheduler runs if necessary when the ISR is complete. It is mandatory to use either these macros or the dispatcher for any interrupt service routine that makes a DSP/BIOS scheduling call such as SWI\_post, SEM\_post, or TSK\_yield.

1. Choose Project → Add Files to Project. Select dma\_isr.s62 and click Open.
2. In the Project View, double-click on dma\_isr.s62. Notice the calls to HWI\_enter and HWI\_exit and the branches to the original C ISRs.
3. Choose File → Close.

#### **Step E: Alter the configuration file so that the assembly routines handle 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. In the function box, replace \_DMARxCisr with DMArxAisr. The underscore is no longer necessary because the new ISR functions are written in assembly. Click OK.
2. Right-click on HWI\_INT9 and choose Properties from the pop-up menu. In the function box, replace \_DMAtxCisr with DMAtxAisr. Click OK.
3. Choose File → Save. Close the Configuration Tool by choosing File → Close.

#### **Step F: Run the program.**

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

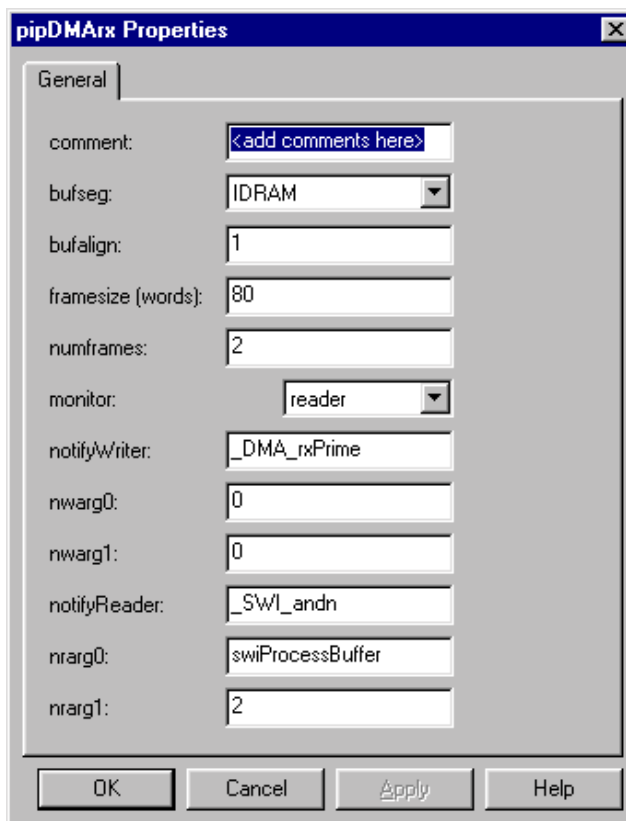
## Appendix A. Incorporating Pipes into the Audio Application

This appendix introduces writing a Buffered Pipe (PIP module) driver for the audio application. The Buffered Pipe Manager provides a convenient interface to peripheral devices such as analog-to-digital (A/D) or digital-to-analog (D/A) converters.

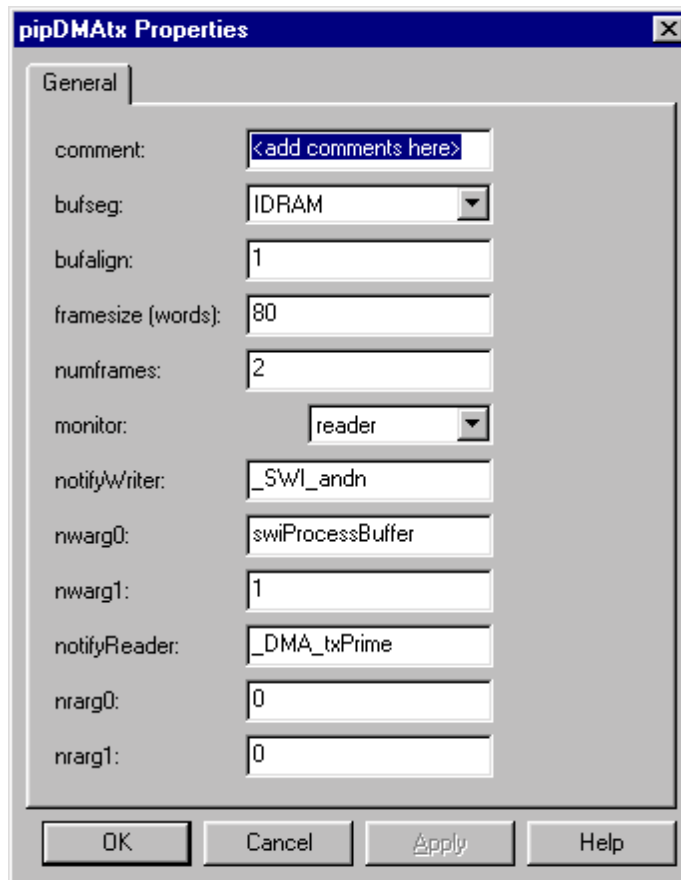
### Step A: Incorporate Pipes into the program.

Instead of using the ping-pong buffers from the original program, you can use pipes. The pipes buffer the data for you.

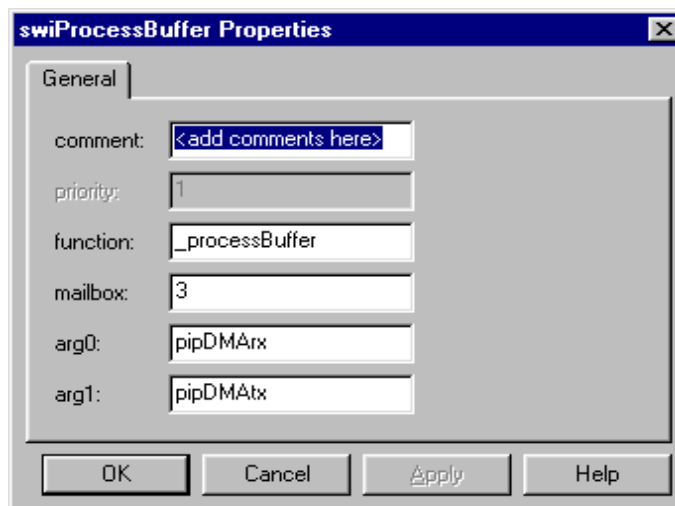
1. In C:\ti\myprojects\biosbydegrees, create a new folder called AudioPIP. Copy all the files From Degree3 into AudioPIP.
2. Double-click on the Code Composer Studio icon on the desktop. Choose Project → Open and browse to the AudioPIP folder. Select the audio.mak and open the project.
3. Click on the + sign next to the Project folder in the Project View and then click on the + sign next to audio.mak. Click on the + sign next to the Source folder and the DSP/BIOS Config folder.
4. Double-click on audio.cdb. Right-click on the Buffered Pipe Manager and choose Insert PIP from the pop-up menu. Right-click on the new PIP object, PIP0, and choose Rename from the pop-up menu. Rename the object pipDMARx.
5. Right-click on pipDMARx and choose Properties. Set the following properties for pipDMARx. When you are done, click OK.



6. Right-click on the Buffered Pipe Manager and choose Insert PIP. Right-click on the new PIP object and choose Rename from the pop-up menu. Rename the object pipDMAtx.
7. Right-click on pipDMAtx and choose Properties from the pop-up menu. Set the following properties for pipDMAtx. When you are done, click OK.



8. Alter swiProcessBuffer so that the software interrupt posts as a result of the pipes' notifyReader() and notifyWriter() function. Click on the + sign next to the Software Interrupt Manager. Right-click on swiProcessBuffer and choose properties from the pop-up menu. Change the properties according to the example below and click OK.



The two PIP objects, pipDMArx and pipDMAtx, are the parameters to the modified processBuffer() function. (You modify processBuffer() in Step B). The mailbox value acts as a mask which has bits cleared as a result of pipDMAtx's notifyWriter function and pipDMArx's notifyReader function. When the mailbox value is cleared to zero, processBuffer() runs and the mailbox value is reset to 3. (Look at the properties of the two PIP objects and notice the calls to SWI\_andn. Consult Help or the DSP/BIOS User's Guide for more information on notifyReader, notifyWriter, and SWI\_andn).

9. Choose File → Save. Close the Configuration Tool.

### Step B: Copy Data from pipDMArx to pipDMAtx.

Now that the program uses pipes instead of the ping-pong buffers, the processBuffer() routine needs to move the data from the pipDMArx to the pipDMAtx.

1. Double-click on audio.c. Add pip.h to the file inclusions. Remember to include pip.h after std.h.
2. Change the processBuffer() function to the following. (You can copy and paste the source code from this document.) The code in this and the following examples incorporates DSP/BIOS data types. The DSP/BIOS API does not use C types. This is in order to ensure portability to other processors that support DSP/BIOS. To consult a complete list of DSP/BIOS data types, refer to the DSP/BIOS User's Guide. Remember to replace the block characters in the for loop with minus signs.

```

/*
 * ===== processBuffer =====
 */
void processBuffer(PIP_Obj *in, PIP_Obj *out)
{
    Uns *src, *dst;
    Uns size;

    if (PIP_getReaderNumFrames(in) == 0 || PIP_getWriterNumFrames(out) == 0)
    {
        error();
    }

    /* get input data and allocate output buffer */
    PIP_get(in);
    PIP_alloc(out);
}

```

```

/* copy input data to output buffer */
src = PIP_getReaderAddr(in);
dst = PIP_getWriterAddr(out);

size = PIP_getReaderSize(in);
PIP_setWriterSize(out,size);

for (; size > 0; size--)
{
    *dst++ = *src++;
}

/* output copied data and free input buffer */
PIP_put(out);    /* Runs NotifyReader */
PIP_free(in);    /* Runs NotifyWriter */
}

```

3. Remove the processBuffer() definition from the beginning of the audio.c file.
4. Declare the error() function by adding the following line before main():

```
static void error(void);
```

5. Add the error() function. Add the following code to the audio.c file. (This function can be copied and pasted from this document.)

```

/*
 * ===== error =====
 */
static void error(void)
{
    LOG_printf(&trace, "Error: audio signal falsely triggered!");

    for (;;)
    {
        ; /* loop forever */
    }
}

```

6. Remove the calls to DMA\_rxStart() and DMA\_txStart() from main(). Starting the DMA explicitly is no longer necessary because the pipes effectively start the DMA for you through the use of their notifyReader() and notifyWriter() functions.
7. Choose File → Save. Close the file.

### Step C: Modify the source code of dma.c to incorporate the pipes.

1. Double-click on dma.c in the Project View. Add pip.h file inclusion. Remember to include pip.h after std.h.
2. Add the following declarations to dma.c.

```

/* Objects created by the Configuration Tool */
extern far PIP_Obj pipDMArx;
extern far PIP_Obj pipDMAtx;

Int DMA_rxCnt = 0;
Int DMA_txCnt = 0;

Int *DMA_rxPtr = NULL;
Int *DMA_txPtr = NULL;

Void DMA_rxPrime(Void);
Void DMA_txPrime(Void);

```

3. Modify DMArxCisr() so that it looks like the following example.

```

/*
 * ===== DMArxCisr =====
 * FUNCTION: Interrupt service routine for DMA channel 0
 */

```



```

Void DMARxCisr(Void)
{
    PIP_put(&pipDMARx);
    DMA_rxCnt = 0;

    /* enable end-of-frame interrupt */
    DMA0_SCNTL = SCNTL_ENABLE;
    DMA_rxPrime();
}
    
```

4. Modify DMAtxCisr() so that it looks like following example.

```

/*
 * ===== DMAtxCisr =====
 * FUNCTION: Interrupt service routine for DMA channel 1
 */
Void DMAtxCisr(Void)
{
    PIP_free(&pipDMAtx);
    DMA_txCnt = 0;

    /* Enable end-of-frame interrupt */
    DMA1_SCNTL = SCNTL_ENABLE;
    DMA_txPrime();
}
    
```

5. Add the DMA\_rxPrime() and DMA\_txPrime() functions to dma.c. Remember to replace the block characters in DMA\_txPrime() with minus signs.

```

/*
 * ===== DMA_rxPrime =====
 */
Void DMA_rxPrime(Void)
{
    PIP_Obj *rxPipe = &pipDMARx;
    static Int nested = 0;

    if (nested)
    {
        /* prohibit recursive call via PIP_alloc() */
        return;
    }

    nested = 1;

    if (DMA_rxCnt == 0 && PIP_getWriterNumFrames(rxPipe) > 0)
    {
        PIP_alloc(rxPipe);

        /* must set 'Ptr' before 'Cnt' to synchronize with isr()*/
        DMA_rxPtr = PIP_getWriterAddr(rxPipe);
        DMA_rxCnt = PIP_getWriterSize(rxPipe);

        DMA_rxStart((Uns)DMA_rxPtr, DMA_rxCnt);
    }

    nested = 0;
}

/*
 * ===== DMA_txPrime =====
 */
Void DMA_txPrime(Void)
{
    PIP_Obj *txPipe = &pipDMAtx;
    static Int delay = 1; /* or 2, 3, etc. */
    static Int nested = 0;

    if (nested)
    {
        /* prohibit recursive call via PIP_get() */
        return;
    }

    if (delay)
    {
    
```

```

        delay--;
        return;
    }
    nested = 1;
    if (DMA_txCnt == 0 && PIP_getReaderNumFrames(txPipe) > 0)
    {
        PIP_get(txPipe);

        /* must set 'Ptr' before 'Cnt' to synchronize with isr()*/
        DMA_txPtr = PIP_getReaderAddr(txPipe);
        DMA_txCnt = PIP_getReaderSize(txPipe);

        DMA_txStart((Uns)DMA_txPtr, DMA_txCnt);
    }
    nested = 0;
}

```

### Step D: Change the DMA from autoinitialized to non–autoinitialized.

Instead of continuously processing frames, the DMA now depends upon the pipe's notifyReader() and notifyWriter() functions before transferring data from the McBSP to the receive pipe, and before transferring from the transmit pipe back to the McBSP. Signaling with the notifyWriter() and notifyReader() functions ensures that pipe buffers are available before the DMA proceeds.

1. In the DMA\_rxStart() and DMA\_txStart() functions (in dma.c), change DMA\_START\_AUTO to DMA\_START.
2. Choose File → Save. Close the file.
3. In the Project View, click on the + sign next to the Include folder. Double-click on AudioDMA.h. Change the defined value FRAMES\_PER\_BLOCK from 0x00020000 to 0x00010000. This value is stored into the DMA channels' transfer count registers. The high nibble of the transfer count register indicates how many frames to transfer per block. Now that the DMA is not autoinitialized, it transfers one block at a time instead of two.
4. Choose File → Save and close the file. Click on the – sign next to the Include folder to collapse the list of header files.
5. In the Project View, double-click on init.c. Scroll down to the enableDMA() function. Change PCNTL\_ENABLE\_AUTO\_0 to PCNTL\_ENABLE\_0.
6. Change PCNTL\_ENABLE\_AUTO\_1 to PCNTL\_ENABLE\_1. This changes the settings of the DMA primary control registers so that the DMA is no longer autoinitialized.
7. Choose File → Save. Close the file.

### Step E: Run the program.

1. Choose Project → Build. Load the new executable.
2. Choose Debug → Go Main and then Debug → Run.

## Appendix B. Source Code Listings for Zero Degree – non-BIOS Audio Application

```

- - - - - File: Audio.c - - - - -
/*
 * =====
 *      DSP/BIOS by Degrees
 *
 * ===== audio.c =====
 *
 */
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "AudioDMA.h"

unsigned int ping_RX[BUFSIZE];
unsigned int ping_TX[BUFSIZE];
unsigned int pong_RX[BUFSIZE];
unsigned int pong_TX[BUFSIZE];

volatile int dataReadyFlag = 0;

void processBuffer(void);
void generateSine(void);

/*
 * ===== main =====
 */
void main()
{
    int date = 25;
    float percent = 10.26;

    printf("Why do computer scientists celebrate Halloween and Christmas on the same day?\n
        \nBecause oct %o equals dec %d. (%6.2f percent of the population gets this joke.)\n",
        date, date, percent);

    initApplication();
    initInterrupts();

    /* Start DMA with autoinitialization */
    DMA_rxStart((unsigned int)&ping_RX[0], DMA_FRAME_SIZE);
    DMA_txStart((unsigned int)&ping_TX[0], DMA_FRAME_SIZE);

    printf("DMA started.\n");

    while(1)
    {
        if(dataReadyFlag)
        {
            dataReadyFlag = 0;
            processBuffer();
        }
    }
}

/*
 * ===== processBuffer =====
 * Copy full RX buffer to available TX buffer using memcpy()
 */
void processBuffer(void)
{
    extern int ping_tx_flag;
    extern int ping_rx_flag;
    char *src, *dst;

    if(ping_rx_flag == 1)
        src = (char *)&pong_RX[0];
    else
        src = (char *)&ping_RX[0];
}

```

```

    if(ping_tx_flag == 1)
        dst = (char *)&pong_TX[0];
    else
        dst = (char *)&ping_TX[0];

    memcpy(dst, src, DMA_FRAME_SIZE*sizeof(unsigned int));
}
/*
 * ===== 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;
    }
}
- - - - - File: dma.c - - - - -
/*
 * ===== dma.c =====
 */
#include "AudioDMA.h"
extern int dataReadyFlag;
int ping_rx_flag = 1;
int ping_tx_flag = 0;
unsigned int dst, src;
/*
 * ===== DMA_rxStart =====
 * Starts DMA channel 0 with autoinitialization
 */
void DMA_rxStart(unsigned int dst, int count)
{
    DMA0_SRC = DRR; /* set source address */
    DMA0_DST = dst; /* set dst address */
    DMA0_TXCNT = FRAMES_PER_BLOCK + count; /* set transfer count */
    DMA0_PCNTL |= DMA_START_AUTO; /* start block transfer */
}
/*
 * ===== DMA_txStart =====
 * Starts DMA channel 1 with autoinitialization
 */
void DMA_txStart(unsigned int src, int count)
{
    DMA1_SRC = src; /* set src address */
    DMA1_DST = DXR; /* set dst address to DXR */
    DMA1_TXCNT = FRAMES_PER_BLOCK + count; /* set transfer count */
    DMA1_PCNTL |= DMA_START_AUTO; /* start block transfer */
}
/*
 * ===== DMARxCisr =====
 * Interrupt service routine for DMA channel 0
 */
interrupt void DMARxCisr(void)
{
    /* enable end-of-frame interrupt */
    DMA0_SCNTL = SCNTL_ENABLE;

    /* buffer is available for processing */
    dataReadyFlag = 1;
}

```

```

    /* ping_rx_flag starts at 1 and toggles with each interrupt */
    if(ping_rx_flag == 1)
        ping_rx_flag = 0;
    else
        ping_rx_flag = 1;
}
/*
 * ===== DMAtxCisr =====
 * Interrupt service routine for DMA channel 1
 */
interrupt void DMAtxCisr(void)
{
    /* Enable end-of-frame interrupt */
    DMA1_SCNTL = SCNTL_ENABLE;

    /* ping_tx_flag starts at 0 and toggles with each interrupt */
    if(ping_tx_flag == 1)
        ping_tx_flag = 0;
    else
        ping_tx_flag = 1;
}

- - - - - File: AudioDMA.h - - - - -

/*
 * ===== AudioDMA.h =====
 */
#ifndef AUDIODMA_
#define AUDIODMA_

#define DMA_FRAME_SIZE  0x0400
#define BUFSIZE         0x500
#define FRAMES_PER_BLOCK 0x00020000

#define PI 3.1415927

/* Macros */
#define SETBITMASKCODEC(i,n)\
        IAR = (i);\
        IDR |= (n);

#define SETREGCODEC(i,n)\
        IAR = (i);\
        IDR = (n);

/* Control Registers */
extern cregister volatile unsigned int CSR;    /* Control Status Register */
extern cregister volatile unsigned int IER;    /* Interrupt Enable Register */

/* Memory Map 1 */

/* DMA Global Address Registers */
#define DMA_GARA    (*(volatile unsigned int *)0x01840038)
#define DMA_GARB    (*(volatile unsigned int *)0x0184003C)
#define DMA_GARC    (*(volatile unsigned int *)0x01840068)
#define DMA_GARD    (*(volatile unsigned int *)0x0184006C)
#define DMA_GARR    (*(volatile unsigned int *)0x01840028)
#define DMA_GRRB    (*(volatile unsigned int *)0x0184002C)

/* DMA Global Index Registers */
#define DMA_GIRA    (*(volatile unsigned int *)0x01840030)
#define DMA_GIRB    (*(volatile unsigned int *)0x01840034)

/* DMA0 Memory Mapped Registers */
#define DMA0_PCNTL (*(volatile unsigned int *)0x01840000) /* Pri cntl Reg */
#define DMA0_SCNTL (*(volatile unsigned int *)0x01840008) /* Sec cntl Reg */
#define DMA0_SRC    (*(volatile unsigned int *)0x01840010) /* Src Addr Reg */
#define DMA0_DST    (*(volatile unsigned int *)0x01840018) /* Dst Addr Reg */
#define DMA0_TXCNT  (*(volatile unsigned int *)0x01840020) /* Transf Cnt Reg */

/* DMA1 Memory Mapped Registers */
#define DMA1_PCNTL (*(volatile unsigned int *)0x01840040) /* Pri cntl Reg */
#define DMA1_SCNTL (*(volatile unsigned int *)0x01840048) /* Sec cntl Reg */

```

```

#define DMA1_SRC    (*(volatile unsigned int *)0x01840050) /* Src Addr Reg */
#define DMA1_DST    (*(volatile unsigned int *)0x01840058) /* Dst Addr Reg */
#define DMA1_TXCNT  (*(volatile unsigned int *)0x01840060) /* Transf Cnt Reg */

/* DMA Commands */
#define PCNTL_STOP      0x00          /* Stop DMA channel */
#define PCNTL_ENABLE_0  0x02034040   /* No auto-init, dst++, SP 0 rx int */
#define PCNTL_ENABLE_1  0x02600010   /* No auto-init, src++, SP 0 tx int */
#define PCNTL_ENABLE_AUTO_0 0x420340C0 /* Auto-init */
#define PCNTL_ENABLE_AUTO_1 0x22036030 /* Auto-init */
#define SCNTL_ENABLE    0x08          /* Enable end-of-frame interrupt */
#define SCNTL_BLOCK_ENABLE 0x80       /* Enable end-of-block interrupt */
#define DMA_START      0x01          /* Start without auto-init */
#define DMA_START_AUTO 0x03          /* Start with auto-init */

/* DMA Interrupts */
#define DMA0_RXINT_BIT 0x0100        /* DMA0 rx interrupt on INT 8 */
#define DMA1_TXINT_BIT 0x0200        /* DMA1 tx interrupt on INT 9 */

/* McBSP 0 Memory Mapped Registers */
#define DRR  0x018c0000 /* McBSP0 rx register */
#define DXR  0x018c0004 /* McBSP0 tx register */
#define SPCR (*(volatile unsigned int *)0x018c0008) /* Serial Port Cont Reg */
#define RCR  (*(volatile unsigned int *)0x018c000c) /* Receive Control Reg */
#define XCR  (*(volatile unsigned int *)0x018c0010) /* Transmit Control Reg */
#define MCR  (*(volatile unsigned int *)0x018c0018) /* Multichannel Reg */
#define PCR  (*(volatile unsigned int *)0x018c0024) /* Pin Control Reg */

/* CPLD (Complex Programmable Logic Device) Register */
#define CPLD (*(volatile unsigned int *)0x01780000)

/* Codec Memory Mapped Registers */
#define IAR  (*(volatile unsigned int *)0x01720000) /* Index Address Reg */
#define IDR  (*(volatile unsigned int *)0x01720004) /* Indexed Data Reg */
#define SR   (*(volatile unsigned int *)0x01720008) /* Status Reg */
#define PIO  (*(volatile unsigned int *)0x0172000c) /* PIO Data Reg */

/* Codec Indirect Mapped Registers */
#define DAC_LEFT_CNTL  0x06 /* I6 - Left DAC Output Control */
#define DAC_RIGHT_CNTL 0x07 /* I7 - Right DAC Output Control */
#define FS_PDF_CNTL    0x48 /* I8 - FS & Playback Data Control (MCE set) */
#define PIO_CNTL       0x49 /* I9 - Interface Control (MCE set) */
#define MODE_CNTL      0x0c /* I12 - Mode and ID Control */
#define SP_CNTL        0x50 /* I16 - Alternate Feature enable 1 (MCE set) */
#define LINE_LEFT_CNTL 0x12 /* I18 - Left Line Input Control */
#define LINE_RIGHT_CNTL 0x13 /* I19 - Right Line Input Control */
#define CDF_CNTL       0x5c /* I28 - Capture Data Format (MCE set) */

/* Codec Commands */
#define DAC_UNMUTE  0x00 /* Unmute DAC-to-mixer */
#define MODE2_ENABLE 0x40 /* Enable MODE 2 */
#define INIT_DONE   0x80 /* Initialization bit */
#define SP_32_ENABLE 0x0a /* Serial Port (32 bits) */
#define SP_64E_ENABLE 0x02 /* Serial Port (64 bits enhanced) */
#define SP_64_ENABLE 0x06 /* Serial Port (64 bits enhanced) */
#define CDF_S_L16    0x50 /* CDF; Stereo, linear, 16-bit */
#define FS8_S_L16    0x50 /* FS: 8kHz, PDF; Stereo, linear, 16-bit */
#define FS44_S_L16   0x5c /* FS: 44kHz, PDF; Stereo, linear, 16-bit */
#define PIO_ENABLE   0xc3 /* Enable PIO (capture and playback) */
#define MCE_RESET    0xbf /* Reset Mode Control Bit (MCE) */

/* CPLD Commands */
#define CODEC_DISABLE 0xf8
#define CODEC_ENABLE  0x04

/* McBSP 0 Commands */
#define SPCR_DISABLE  0x00 /* Disable serial port */
#define SPCR_STANDBY  0x00200020 /* Wait for frame synch, enable RX int */
#define SPCR_ENABLE   0x00010001 /* enable serial port (interrupt CPU) */
#define SPCR_ENABLE_DMA 0x00110011 /* enable serial port (!interrupt CPU) */
#define PCR_SET       0x00 /* external RX/TX clock and frame synch */
#define CR_32_SET     0x000000a0 /* 32 bit word, 1 w/frame, single phase */
#define CR_SET        0x000101a0 /* 32 bit word, 2 w/frame, single phase */
#define MCR_SET       0x00 /* enable TX and RX channels */
#define MCSP_RXINT_BIT 0x0800 /* assume serial RX interrupt on INT 11 */

```

```

#endif
- - - - - File: init.c - - - - -
/*
 * ===== init.c =====
 */
#include "AudioDMA.h"
static void enableCodec(void); /* Configure Codec */
static void enableMcBSP0(void); /* Configure McBSP 0 */
static void enableDMA(void); /* Configure McBSP 0 to use DMA */
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int ICR;
extern cregister volatile unsigned int IER;
extern unsigned int ping_RX[BUFSIZE];
extern unsigned int pong_RX[BUFSIZE];
extern unsigned int ping_TX[BUFSIZE];
extern unsigned int pong_TX[BUFSIZE];
/*
 * ===== initApplication =====
 * Initialize buffers and peripherals
 */
void initApplication(void)
{
    int index;

    /* Initialize buffers */
    for(index=0; index < BUFSIZE; index++)
    {
        ping_RX[index] = 0x00000000;
        pong_RX[index] = 0x00000000;
        ping_TX[index] = 0x00000000;
        pong_TX[index] = 0x00000000;
    }

    /* Configure Codec */
    enableCodec();

    /* Configure McBSP 0 */
    enableMcBSP0();

    /* Configure DMA */
    enableDMA();

    /* Unmute left/right DAC-to-mixer */
    SETREGCODEC(DAC_LEFT_CNTL, DAC_UNMUTE);
    SETREGCODEC(DAC_RIGHT_CNTL, DAC_UNMUTE);
}
/*
 * ===== initInterrupts =====
 */
void initInterrupts(void)
{
    /* enable NMIE */
    IER |= 0x00000002;

    /* enable GIE */
    CSR |= 0x00000001;
}
/*
 * ===== enableCodec =====
 */
static void enableCodec(void)
{
    /* Reset and enable codec */
    CPLD &= CODEC_DISABLE;
    CPLD |= CODEC_ENABLE;

    /* wait for codec to finish initialization */
    while(IAR & INIT_DONE);
}

```

```

/* Enable mode 2 */
SETBITMASKCODEC(MODE_CNTL, MODE2_ENABLE);

/* Set codec Serial port format */
SETREGCODEC(SP_CNTL, SP_32_ENABLE);

/* Set capture format */
SETREGCODEC(CDF_CNTL, CDF_S_L16);

/* Set playback format and sample rate */
SETREGCODEC(FS_PDF_CNTL, FS8_S_L16);

/* Enable bits in PIO (programmed I/O) */
SETREGCODEC(PIO_CNTL, PIO_ENABLE);

/* Reset Mode Control Bit (MCE) */
IAR &= MCE_RESET;
}

/*
 * ===== enableMcBSP0 =====
 */
static void enableMcBSP0(void)
{
    /* Disable serial port */
    SPCR = SPCR_DISABLE;

    /* Configure pin control register */
    PCR = PCR_SET;

    /* Configure receive control register */
    RCR = CR_32_SET;

    /* Configure transmit control register */
    XCR = CR_32_SET;

    /* Enable transmit and receive channels */
    MCR = MCR_SET;

    /* Put serial port in 'standby' and wait for frame synch */
    SPCR = SPCR_STANDBY;

    /* Clear the frame synch interrupt */
    ICR = MCSP_RXINT_BIT;

    /* Enable serial port */
    SPCR = SPCR_ENABLE_DMA;
}

/*
 * ===== enableDMA =====
 */
static void enableDMA(void)
{
    /* Clear DMA R/W STAT bits */
    DMA0_SCNTL = 0x00;
    DMA1_SCNTL = 0x00;

    /* Set DMA Primary Control Register */
    DMA0_PCNTL = PCNTL_STOP;
    DMA0_PCNTL = PCNTL_ENABLE_AUTO_0;
    DMA1_PCNTL = PCNTL_STOP;
    DMA1_PCNTL = PCNTL_ENABLE_AUTO_1;

    /* Set DMA Secondary Control Register */
    DMA0_SCNTL = SCNTL_ENABLE; /* end-of-frame interrupt */
    DMA1_SCNTL = SCNTL_ENABLE; /* end-of-frame interrupt */

    /* Set DMA Global Registers for Autoinitialization */
    DMA_GIRA = (unsigned int)&pong_RX[0]
        - (((unsigned int)&ping_RX[0] + (DMA_FRAME_SIZE * sizeof(unsigned int)))
        + sizeof(unsigned int));
    DMA_GIRA = (DMA_GIRA << 16) + sizeof(unsigned int);
    DMA_GIRB = (unsigned int)&pong_TX[0]

```



```

        - ((unsigned int)&ping_TX[0] + (DMA_FRAME_SIZE * sizeof(unsigned int)))
        + sizeof(unsigned int);
DMA_GIRB = (DMA_GIRB << 16) + sizeof(unsigned int);
DMA_GARB = (unsigned int)&ping_RX[0]; /* Reload value for DMA0 */
DMA_GARC = (unsigned int)&ping_TX[0]; /* Reload value for DMA1 */
DMA_GRRR = 0x20000 + DMA_FRAME_SIZE; /* two frames + count per frame */

/* Reset SRC and DST address registers */
DMA0_SRC = 0x00;
DMA0_DST = 0x00;
DMA1_SRC = 0x00;
DMA1_DST = 0x00;

/* Enable DMA-to-CPU interrupts */
IER |= DMA0_RXINT_BIT; /* INT 8 */
IER |= DMA1_TXINT_BIT; /* INT 9 */
}

```

----- File: Vectors.asm -----

```

;
;
; ===== vectors.asm =====
; Plug in the entry point at RESET in the interrupt vector table
; Plug in address of ISR for interrupts 8 and 9
;
;
; ===== unused =====
; plug infinite loop -- with nested branches to
; disable interrupts -- for all undefined vectors
        .ref          _c_int00      ; reset ISR
        .ref          _DMArxCisr   ; ISR for interrupt 8
        .ref          _DMAtxCisr   ; ISR for interrupt 9
        .global       _istb        ; interrupt service table base

```

unused .macro id

.global unused:id:

unused:id:

```

b unused:id:      ; nested branches to block interrupts
nop 4
b unused:id:
nop
nop
nop
nop
nop
nop
nop

```

.endm

.sect ".vec"

```

_istb:
_RESET:          mvkl          _c_int00,b0
                 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          _DMArxCisr,b0
                 mvkh          _DMArxCisr,b0
                 b              b0
                 nop           5

```

```

        nop
        nop
        nop
        nop
_INT9:   mvkl      _DMAtxCisr,b0
        mvkh      _DMAtxCisr,b0
        b         b0
        nop      5
        nop
        nop
        nop
        nop

_INT10:  unused 10
_INT11:  unused 11
_INT12:  unused 12
_INT13:  unused 13
_INT14:  unused 14
_INT15:  unused 15
- - - - - File: Link.cmd - - - - -
/* linker command file for audio */
MEMORY
{
  INT_PROG_MEM (RX)      : origin = 0x00000000 length = 0x00010000
  SBSRAM_PROG_MEM (RX)  : origin = 0x00400000 length = 0x00014000
  SBSRAM_DATA_MEM (RW)  : origin = 0x00414000 length = 0x0002C000
  SDRAM0_DATA_MEM (RW)  : origin = 0x02000000 length = 0x00400000
  SDRAM1_DATA_MEM (RW)  : origin = 0x03000000 length = 0x00400000
  INT_DATA_MEM (RW)     : origin = 0x80000000 length = 0x00010000
}
SECTIONS
{
  .vec:          load = 0x00000000
  .text:         load = SBSRAM_PROG_MEM
  .const:        load = INT_DATA_MEM
  .bss:          load = INT_DATA_MEM
  .data:         load = INT_DATA_MEM
  .cinit         load = INT_DATA_MEM
  .pinit         load = INT_DATA_MEM
  .stack         load = INT_DATA_MEM
  .far           load = INT_DATA_MEM
  .systemem     load = SDRAM0_DATA_MEM
  .cio          load = INT_DATA_MEM
  sbsbuf        load = SBSRAM_DATA_MEM
                { _SbsramDataAddr = .; _SbsramDataSize = 0x0002C000; }
}
- - - - - File: dmaISR.s62 - - - - -
;
; dmaISR.s62
;

  .include c62.h62
  .include hwi.h62
  .include swi.h62

;   .ref _stsProcessBuffer

  .text
  .global DMARxAisr, DMAtxAisr, _DMAtxCisr, _DMARxCisr

;
; ===== DMARxAisr =====
;
DMARxAisr:
    HWI_enter C62_ABTEMPS, C62_CTEMPS, 0xFFFF, C62_PCC_DISABLE

```

```

        b _DMArxCisr
        mvkl rxDone,b3                ; set return pointer to come back here
        mvkh rxDone,b3
        nop 3

rxDone:
        HWI_exit C62_ABTEMPS, C62_CTEMPS, 0xFFFF, C62_PCC_DISABLE

;
; ===== DMAtxAisr =====
;
DMAtxAisr:
        HWI_enter C62_ABTEMPS, C62_CTEMPS, 0xFFFF, C62_PCC_DISABLE
        b _DMAtxCisr
        mvkl txDone,b3                ; set return pointer to come back here
        mvkh txDone,b3
        nop 3

txDone:
        HWI_exit C62_ABTEMPS, C62_CTEMPS, 0xFFFF, C62_PCC_DISABLE
        .end
    
```

## 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