Texas Instruments

Application Report
SPRA598 - November 1999

An Audio Example Using DSP/BIOS

Shawn Dirksen
Digital Signal Processing Solutions

ABSTRACT

Data transfer is essential for any digital signal processing application. Texas Instruments (TI™) DSP/BIOS kernel provides basic runtime services used for managing data transfer. The DSP/BIOS pipes are used to buffer streams of program input and output data. Data transfer is scheduled through the use of DSP/BIOS software interrupts. These software interrupts, patterned after hardware interrupt routines, are the foundation for structuring DSP/BIOS applications in a prioritized hierarchy of *real-time threads*.

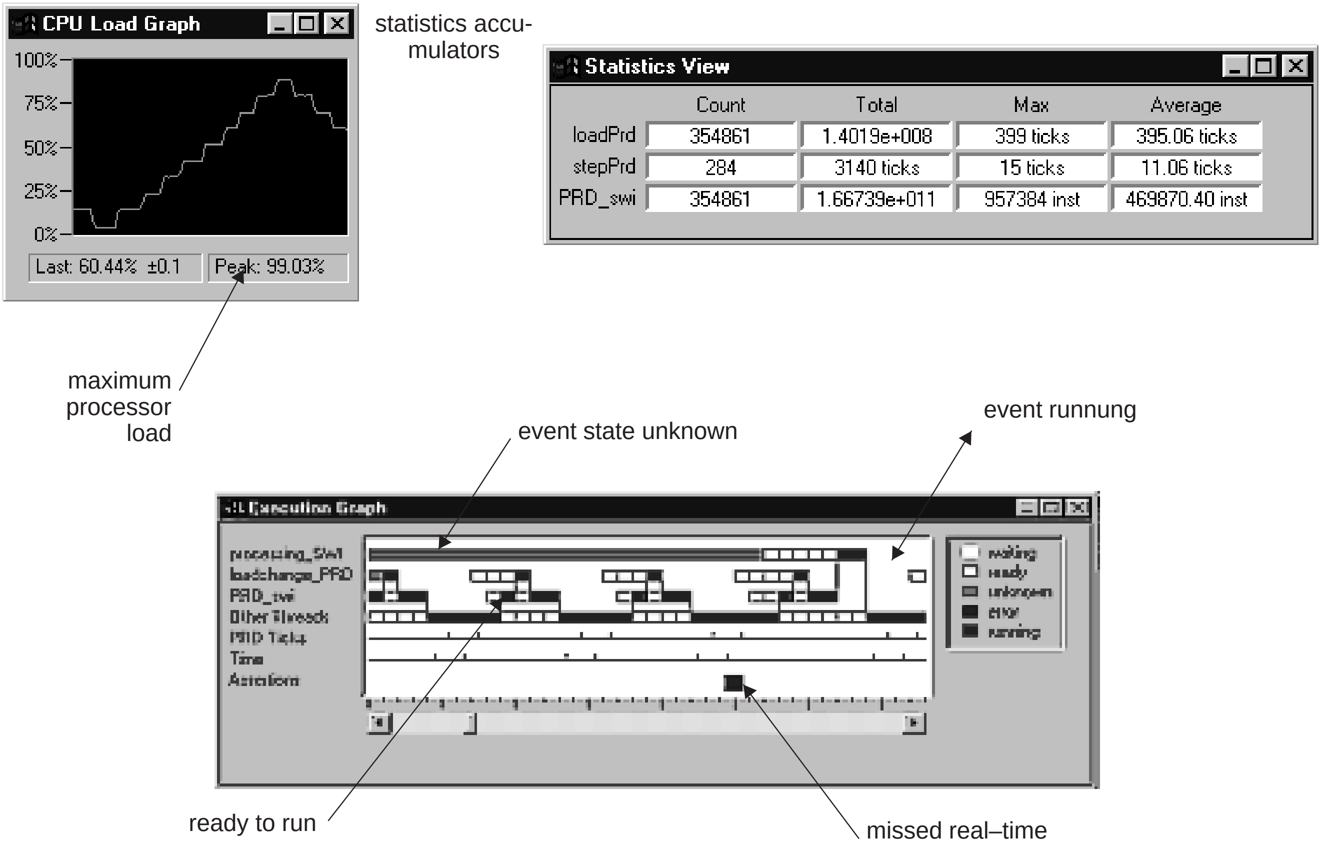
This audio example demonstrates how to use DSP/BIOS APIs for scheduling data transfer between the hardware I/O peripherals and the target DSP.

Contents

List of Figures

1 Introduction to Foundational Software

The ability for digital signal processors to handle high-speed arithmetic, I/O and interrupt processing requires basic scheduling and I/O services. The DSP/BIOS foundation software, included in Code Composer Studio, furnishes a small firmware kernel with basic run-time services that software developers can embed on target DSP hardware. DSP/BIOS includes optimized run-time services such as low-latency threading and scheduling along with a data pipe managers designed to manage block I/O(also called stream-based or asynchronous I/O). The embedded DSP/BIOS run-time library and DSP/BIOS plug-ins support a new generation of testing and diagnostic tools that allows developers and integrators to probe, trace, and monitor a DSP application during its course of execution (see Figure 1. DSP/BIOS Real-Time Analysis Tools) This real-time monitoring lets you view the system running in *real-time* so that you can effectively debug and performance-tune your system before deployment.



Last: 60.44% ±0.1

Peak: 99.03%

statistics accumulators

Statistics View

	Count	Total	Max	Average
loadPrd	354861	1.4019e+008	399 ticks	395.06 ticks
stepPrd	284	3140 ticks	15 ticks	11.06 ticks
PRD_swi	354861	1.66739e+011	957384 inst	469870.40 inst

maximum processor load

event state unknown

event runnung

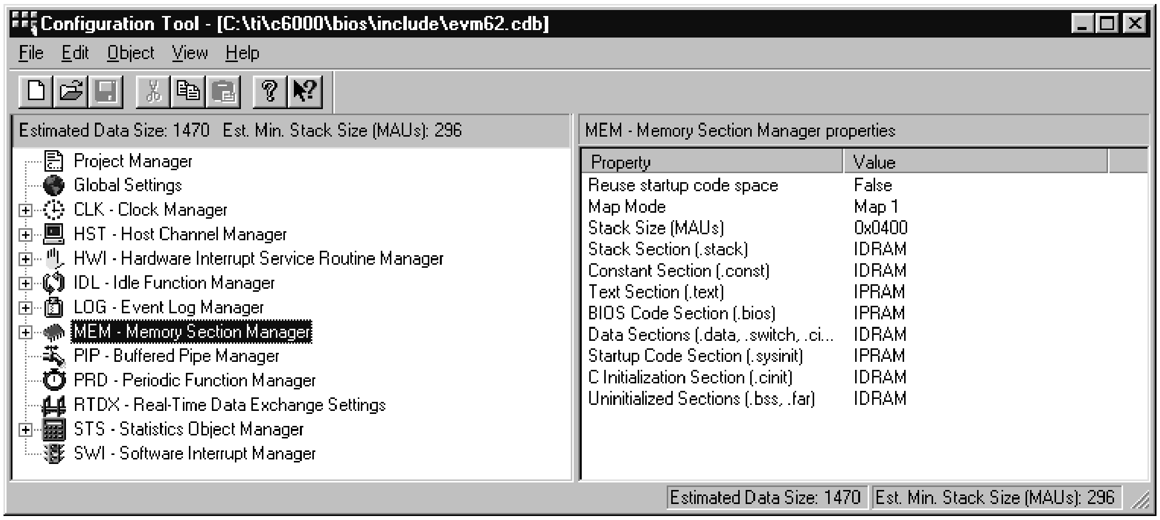
ready to run

missed real–time

Figure 1. DSP/BIOS Real-Time Analysis Tools

Your target application is designed using the DSP/BIOS Configuration Tool for creating and assigning attributes to individual run-time objects (threads, streams, etc.). Unlike other systems in which object creation and initialization occur at run-time thought supplementary API calls, incurring further target overhead –especially code space-, all DSP/BIOS objects are statically configured and bound into an executable program image using hosted tools. In addition to minimizing the target memory footprint by eliminating run-time code and optimizing the layout of internal data structures, the static configuration strategy pursued by the DSP/BIOS Configuration Tool provides the means for early detection of semantic errors through validation of object attributes prior to program execution.

The DSP/BIOS Configuration Tool serves as a visual editor for creating run-time objects that are used on the target application through the DSP/BIOS APIs. This graphical tool makes it easy for a developer to control a wide range of parameters.



Configuration Tool - [C:\ti\c6000\bios\include\evm62.cdb]

Estimated Data Size: 1470 Est. Min. Stack Size (MAUs): 296

- Project Manager
- Global Settings
- CLK - Clock Manager
- HST - Host Channel Manager
- HWI - Hardware Interrupt Service Routine Manager
- IDL - Idle Function Manager
- LOG - Event Log Manager
- MEM - Memory Section Manager
- PIP - Buffered Pipe Manager
- PRD - Periodic Function Manager
- RTDX - Real-Time Data Exchange Settings
- STS - Statistics Object Manager
- SWI - Software Interrupt Manager

MEM - Memory Section Manager properties

Property	Value
Reuse startup code space	False
Map Mode	Map 1
Stack Size (MAUs)	0x0400
Stack Section (.stack)	IDRAM
Constant Section (.const)	IDRAM
Text Section (.text)	IPRAM
BIOS Code Section (.bios)	IPRAM
Data Sections (.data, .switch, .ci...	IDRAM
Startup Code Section (.sysinit)	IPRAM
C Initialization Section (.cinit)	IDRAM
Uninitialized Sections (.bss, .far)	IDRAM

Estimated Data Size: 1470 Est. Min. Stack Size (MAUs): 296

Figure 2. DSP/BIOS Configuration Tool

2 Overview of DSP/BIOS SWI and PIP Modules

The DSP/BIOS kernel is internally organized around a collection of discrete firmware modules, each implementing a coherent subset of the run-time services invoked by the target through kernel APIs. Individual modules in general will manage one or more instances of a related class of kernel objects and will rely upon global parameter values to control their overall behavior, all of which are statically defined using the DSP/BIOS Configuration Tool.

2.1 Software Interrupt or SWI Module

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt routines, are triggered programmatically through DSP/BIOS API calls, such as SWI_post, from client threads. Once triggered, execution of a SWI routine will strictly preempt any current background activity within the program as well as any SWIs of lower priority; HWI hardware interrupt routines on the other hand take precedence over SWIs and remain enabled during execution of all handlers, allowing timely response to hardware peripherals with the target system. Software interrupts or SWIs provide a range of threads that have intermediate priority between HWI functions and the background idle loop.

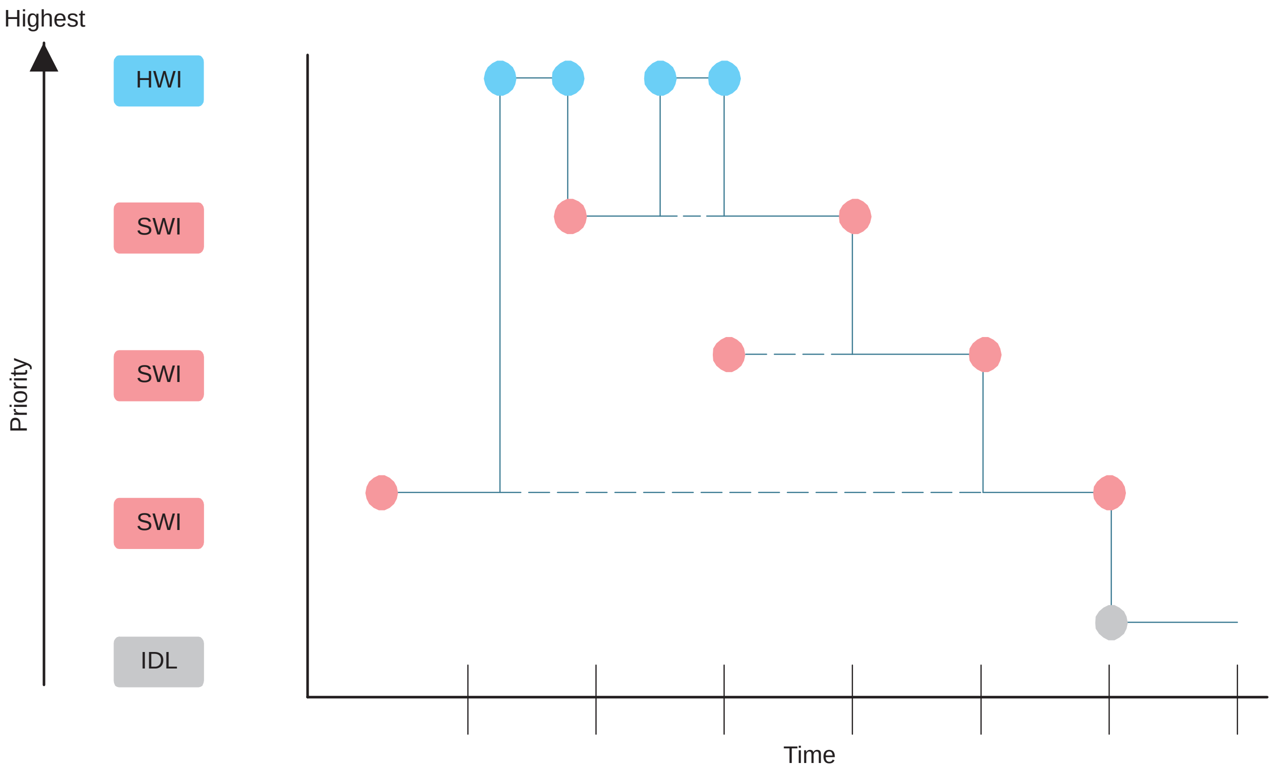


Figure 3. Prioritization of DSP/BIOS Threads

2.2 Pipe or PIP Module

The DSP/BIOS Buffered Pipe Manager or PIP Module manages block I/O (also called stream-based or asynchronous I/O) used to buffer streams of program input and output typically processed by embedded DSP applications. Each pipe object maintains a buffer divided into a fixed number of fixed length frames, specified by the numframes and framesize properties. All I/O operations on a pipe deal with one frame at a time. Although each frame has a fixed length, the application may put a variable amount of data in each frame (up to the length of the frame). Note that a pipe has two ends. The writer end is where the program writes frames of data. The reader end is where the program reads frames of data.

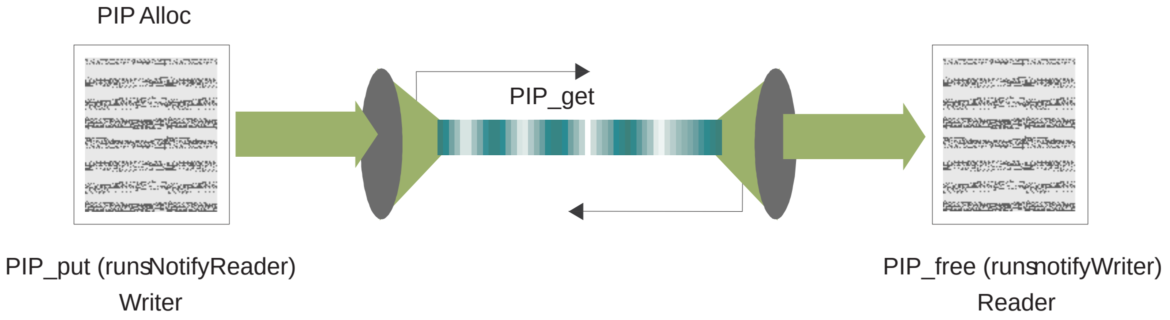


Figure 4. DSP/BIOS Data Pipes (PIP Module)

Data notification functions (notifyReader and notifyWriter) are performed to synchronize data transfer. These functions are triggered when a frame of data is read or written to notify the program that a frame is free or data is available. These functions are performed in the context of the function that calls PIP_free or PIP_put. They may also be called from the thread that calls PIP_get or PIP_alloc. After PIP_alloc is called, DSP/BIOS checks whether there are more full frames in the pipe. If so, the notifyReader function is executed. After PIP_alloc is called, DSP/BIOS whether there are more empty frames in the pipe. If so, the notifyWriter function is executed.

A pipe should have a single reader and a single writer. Often, one end of a pipe is controlled by a hardware ISR (ex: Serial Port Receiver ISR) and on the other end is controlled by a software interrupt function. Pipes can also be used to transfer data within the program between two application threads.

Next, we use two DSP/BIOS Data Pipes in the audio example to move data between the software interrupt function called **audioSWI** and a serial port connected to the codec.

3 An Audio Example

This audio example demonstrates how to use DSP/BIOS APIs for scheduling data transfer between hardware I/O peripherals and the target DSP. This example has been provided to assist with development for your DSP application program. The following steps will guide you through the audio example setup, how to use the DSP/BIOS Configuration Tool and how to use the DSP/BIOS Real-Time Analysis Tools for debugging/testing program code. This example has been installed in both the C5000 or C6000 Code Composer Studio products and can be found in the **...\ti\c6000\examples\bios\audio** or **...\ti\c5400\examples\bios\audio** directories.

3.1 About the Example

The function **audio** is written in C and is located in the source code file **audio.c** found in the **...\ti\c6000\examples\bios\audio** directory. Two pipe objects are used to exchange data between the software interrupt and the serial port connected to the codec. These pipes are **DSS_rxPipe** and **DSS_txPipe**. Data input from the codec flows from the serial port Interrupt Service Routine (ISR) through **DSS_rxPipe** to the software interrupt, where it is copied to **DSS_txPipe** and sent back to the serial port ISR to be transmitted out through the codec, as shown in Figure 5.

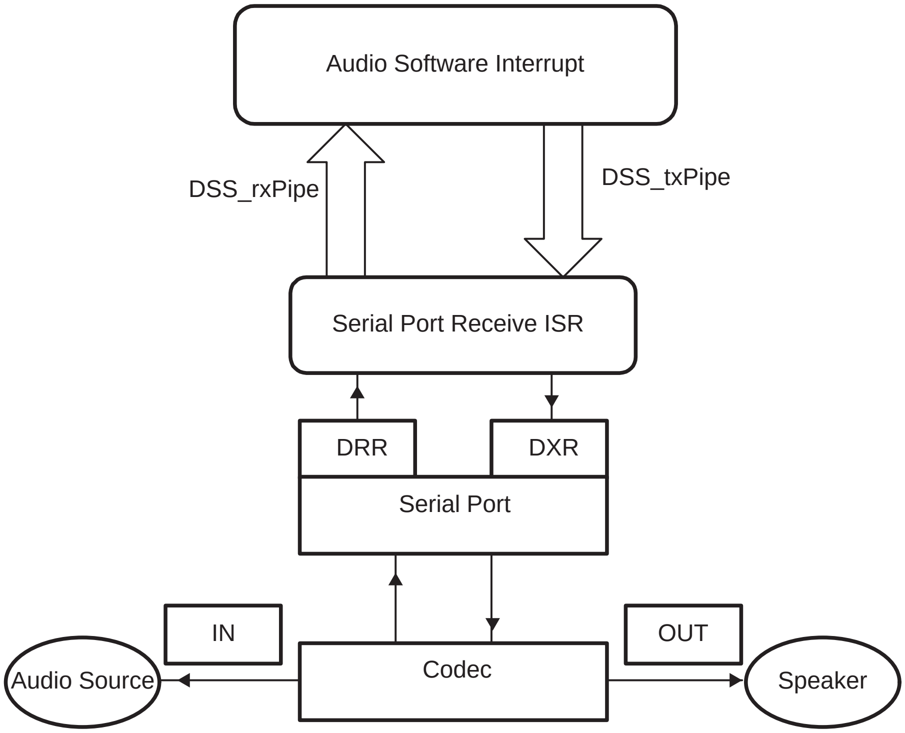


Figure 5. Diagram of the Audio Example

The ISR for the serial port receive interrupt copies each new 32 bit data sample in the Data Receive Register (DRR) to a frame from the **DSS_rxPipe** pipe object. When the frame is full, the ISR puts the frame back into **DSS_rxPipe** to be read by the **audio** function.

As the **audio** function will just read a frame from **DSS_rxPipe** and copy it to a frame in **DSS_txPipe.** The transmit rate will be the same as the receive rate: 48 kHz. This allows us to further simplify the example by enabling *only* the receive interrupt for the serial port. The transmit interrupt for the serial port is not enabled.

The ISR for the receive interrupt will also take care of the transmit process in the serial port.

It will take a full frame from **DSS_txPipe** and write a 32-bit word from the frame to the 32-bit serial port Data Transmit Register (DXR) each time the interrupt is handled. When the whole frame has been transmitted, the empty frame is recycled back to **DSS_txPipe** for reuse by the **audio** function.

The function **DSS_init** (found in **dss.c**) takes care of the initialization of the serial port and the codec. **DSS_init** programs the sampling rate of the codec and sets the bits in IMR and IFR to enable the serial port receive interrupt, etc. Notice the following:

- DSS_init does not enable interrupts and should be called before interrupts are enabled by DSP/BIOS at the return from main.
- DSS_init does not set up the interrupt vector table to call the ISR for the serial port receive interrupt. This is performed and setup with the HWI – Hardware Interrupt Service Routine Manager in the DSP/BIOS Configuration Tool.

3.2 Configuration Setup

All DSP/BIOS objects are pre-configured and bound into an executable program image. This is done through the DSP/BIOS Configuration Tool. When you save a configuration file, the Configuration Tool creates assembly and header files and a linker command file to match your settings. These files that are then linked with your code when building your application program. See the sections “Using the Configuration Tool” in the DSP/BIOS User’s Guide and/or “Creating a Configuration File” in the Code Composer Studio Tutorial for more information.

The following DSP/BIOS objects will be setup/created in this section:

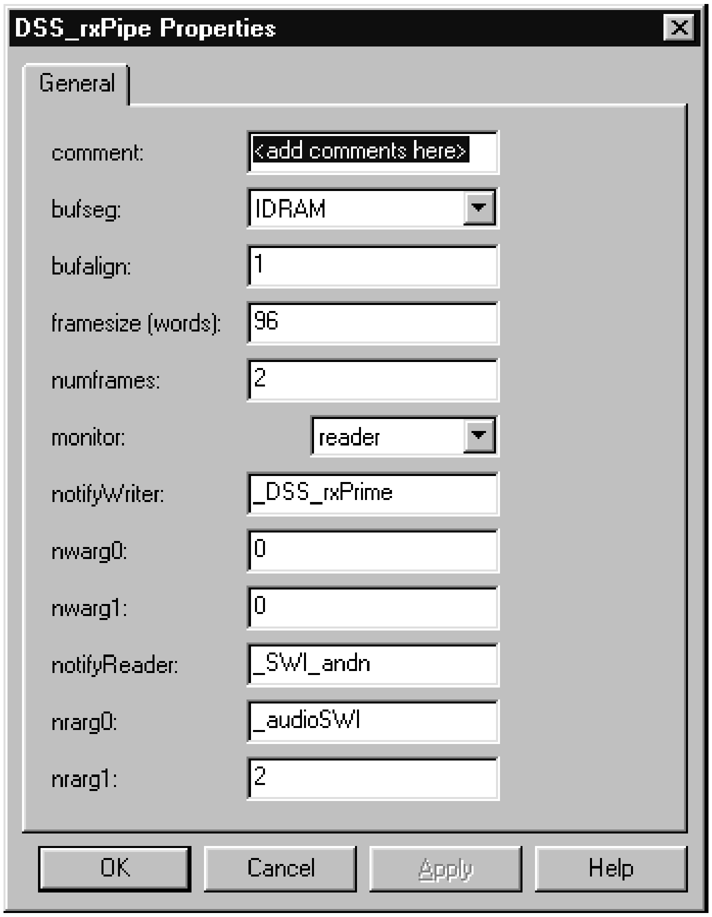
- A software interrupt, **audioSWI**, to run the **audio** function.
- Two data pipes, **DSS_rxPipe** and **DSS_txPipe**, to exchange data between **audioSWI** and the ISR for the serial port receive interrupt.
- Plug in the corresponding ISR for the serial port using the HWI – Hardware Interrupt Service Routine Manager

You begin by opening the project with Code Composer Studio and examining the source code files and libraries used in that project.

1. If you installed Code Composer Studio in **c:\ti**, create a folder called audio in the **c:\ti\myprojects folder**. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)
2. Copy all files from the **c:\ti\c6000\examples\bios\audio** folder to this new folder.
3. From the Windows Start menu, choose Programs –> Code Composer Studio ‘C6000’ –> Code Composer Studio.
4. Choose Project –> New. Type in **audio.mak** for the file name in the folder you created and click Save.
5. Choose File –> New –> DSP/BIOS Configuration.
6. Select the template for your DSP board and click OK.
7. Right-click on the LOG – Event Log Manager and choose the Insert LOG from the pop-up menu. This creates a LOG object called LOG0.
8. Right-click on the name of the LOG0 object and choose Rename from the pop-up menu. Change the object’s name to **trace** and change the buffer length property to **256**.
9. Right-click on the name of the LOG_system object and choose properties. Change the buffer length property to **256**.
10. Right-click on the SWI – Software Interrupt Manager and choose Insert SWI. Rename the new SWI0 object **audioSWI**.
11. Right-click on **audioSWI** and select Properties from the menu. In the **audioSWI** properties window, enter **_audio** for the **function**, **3** for the **mailbox**, **DSS_rxPipe** for **arg0**, and **DSS_txPipe** for **arg1**. Click ***OK*** to save your changes.

12. Right-click on the PIP – Buffered Pipe Manager and choose Insert PIP twice. Rename the first pipe to **DSS_rxPipe** and the second pipe to **DSS_txPipe**.

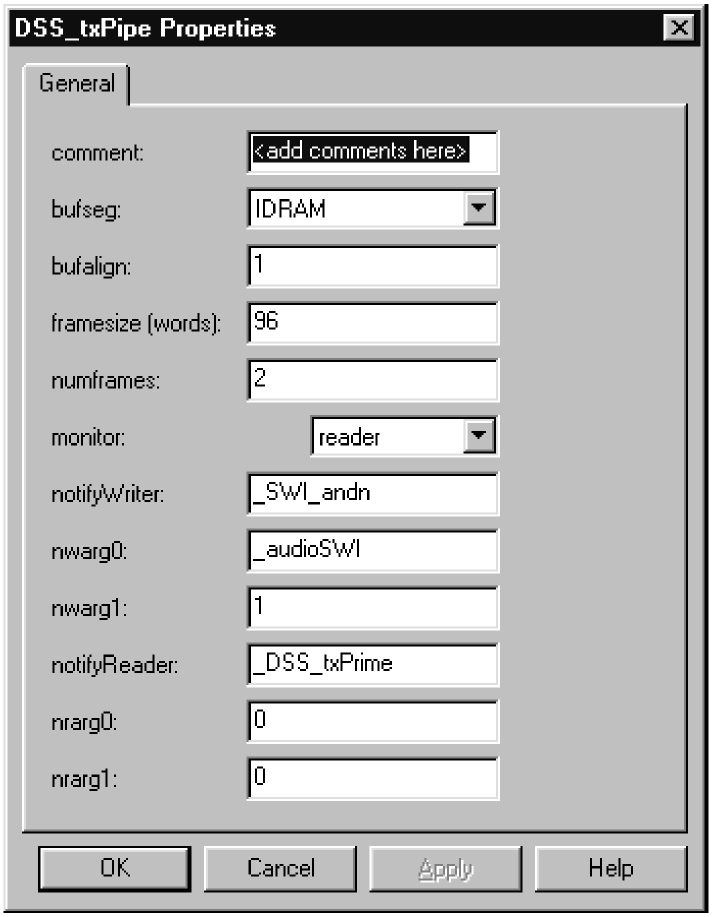
13. Right-click on the **DSS_rxPipe** and select Properties from the menu. Enter the following properties for **DSS_rxPipe**:



comment: <add comments here>
bufseg: IDRAM
bufalign: 1
framesize (words): 96
numframes: 2
monitor: reader
notifyWriter: _DSS_rxPrime
nwarg0: 0
nwarg1: 0
notifyReader: _SWI_andn
nrarg0: _audioSWI
nrarg1: 2

Click ***OK*** to save your changes.

14. Right-click on the **DSS_txPipe** and select Properties from the menu. Enter the following properties for **DSS_txPipe**:



comment: <add comments here>
bufseg: IDRAM
bufalign: 1
framesize (words): 96
numframes: 2
monitor: reader
notifyWriter: _SWI_andn
nwarg0: _audioSWI
nwarg1: 1
notifyReader: _DSS_txPrime
nrarg0: 0
nrarg1: 0

Click ***OK*** to save your changes.

When a full frame is put in **DSS_rxPipe** the **notifyReader** function will clear the second bit in the mailbox for **audioSWI**. When an empty frame is available in **DSS_txPipe**, the first bit in the mailbox for **audioSWI** is cleared. In this way, **audioSWI** is posted only when there is a full frame available in **DSS_rxPipe** and an empty frame available in **DSS_txPipe**.

The **notifyWriter** for **DSS_rxPipe**, **DSS_rxPrime**, is a C function that can be found is **dss.c**. **DSS_rxPrime** calls **PIP_alloc** to allocate an empty frame from **DSS_rxPipe** that will be used by the ISR to write the data received from the codec. **DSS_rxPrime** is called whenever an empty frame is available in **DSS_rxPipe** (and the ISR is done with the previous frame). The ISR calls **DSS_rxPrime** after it is done filling up a frame (see Figure 6)

The **notifyReader** for **DSS_txPipe**, **DSS_txPrime**, is a C function that can be found is **dss.c**. **DSS_txPrime** calls **PIP_get** to get a full frame from **DSS_txPipe**. The data in this frame will be transmitted by the ISR to the codec. **DSS_txPrime** is called whenever a full frame is available in **DSS_txPipe** (and the ISR is done transmitting the previous frame). The ISR calls **DSS_txPrime** after it is done transmitting a frame to get the next full frame. See Figure 6.

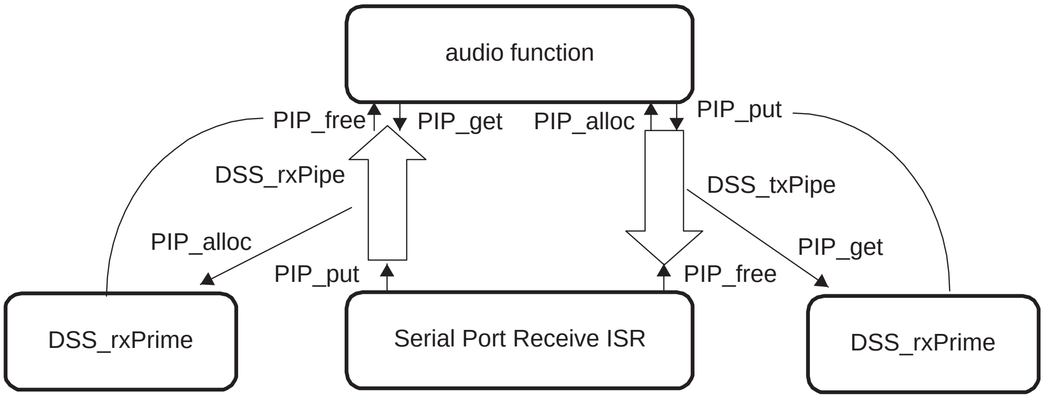
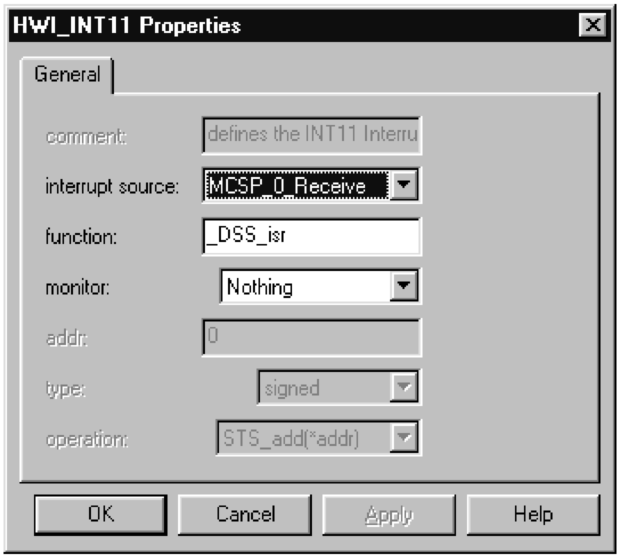


Figure 6. DSS_rxPrime and DSS_txPrime

Now we need to plug in the corresponding ISR for the serial port.

15. Click on the + next to the HWI – Hardware Interrupt Service Routine Manager to display its objects. Each of these objects corresponds to an interrupt in the TMS320C62x interrupt vector table. Right-click on the **HWI_INT11** and select properties. Choose the interrupt source that corresponds to the Multichannel Buffered Serial Port 0 Receive Interrupt (MCSP_0_Receive). Also, change the function to **_DSS_isr** similar to the following:



HWI_INT11 Properties

comment: defines the INT11 Interru
interrupt source: MCSP_0_Receive
function: _DSS_isr
monitor: Nothing
addr: 0
type: signed
operation: STS_add(*addr)

Click ***OK*** to save your changes.

By entering **_DSS_isr** in the function field, DSP/BIOS will set the TMS320C62x interrupt vector table to jump to **_DSS_isr** to handle the serial port receive interrupt.

16. Right-click on the SWI – Software Interrupt Manager object and select Properties from the menu. Select microseconds in the Statistics Units field. This will cause the DSP/BIOS Real-Time Analysis Tools to display the statistic data for the software interrupt in microseconds.
17. Save the configuration file as **audio.cdb**. If asked to replace the existing file; click ***Yes***.
18. Choose Project –> Add Files to Project. Select Configuration File (*.cdb) in the Files of type box. Select the **audio.cdb** file and click Open. Notice that the Project View now contains audio.cdb in a folder called DSP/BIOS Config. In addition, the audiocfg.s62 file is now listed as a source file.
19. The output file name must match the .cdb file name (audio.out and audio.cdb). Go to Project –> Options and choose the Linker tab. Verify the Output Filename field as **audio.out**.
20. Choose Project –> Add Files to Project again. Select Linker Command File (*.cmd) in the Files of type box. Select the **audiocfg.cmd** file and click Open.
21. Choose Project –> Add Files to Project again. Select Source Files (*.c,*s,*.a*) in the Files of type box. Select the **dss.c**, **audio.c**, **dss_evm62.c**, **dss_aisr.s62, audio_ld.s62** files and click Open.
22. Choose Project –> Rebuild All

3.3 Reviewing the Code

Review the code for **audio** in **audio.c** (the ISR will be described at the end of this chapter). Notice that **audio** gets a full frame from **DSS_rxPipe** and an empty frame from **DSS_txPipe**, and copies the contents of the input frame to the output frame. The **audio** function will only run when there is a full frame available in **DSS_rxPipe** and an empty frame available in **DSS_txPipe**.

```
/*
 * ======== audio ========
 */
Void audio(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);
  PIP_free(in);
}
```

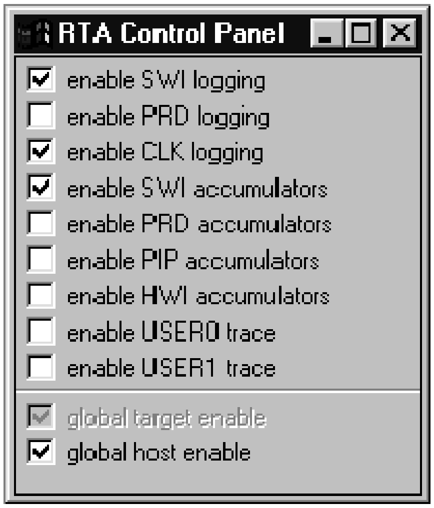
3.4 Debugging and Testing With DSP/BIOS Real-Time Analysis Tools

To run this example you will need to connect the codec input to some acoustical signal. For example, if your PC has a CD-ROM, you can use it to play a CD and connect the earphone’s output of the CD-ROM to the board’s input jack. The output on the board needs to be connected to some output device, e.g., a speaker.

Once the board is connected to an input and output device, start the input signal (e.g., start playing the CD in the CD-ROM).

1. Choose File –> Load Program. Select the program you just built, **audio.out,** and click Open.
2. Choose Debug –> Go Main. The program runs to the first statement in the main function.

3. Choose Tools –> DSP/BIOS –> RTA Control Panel. You see several check boxes at the bottom of the Code Composer Studio window.

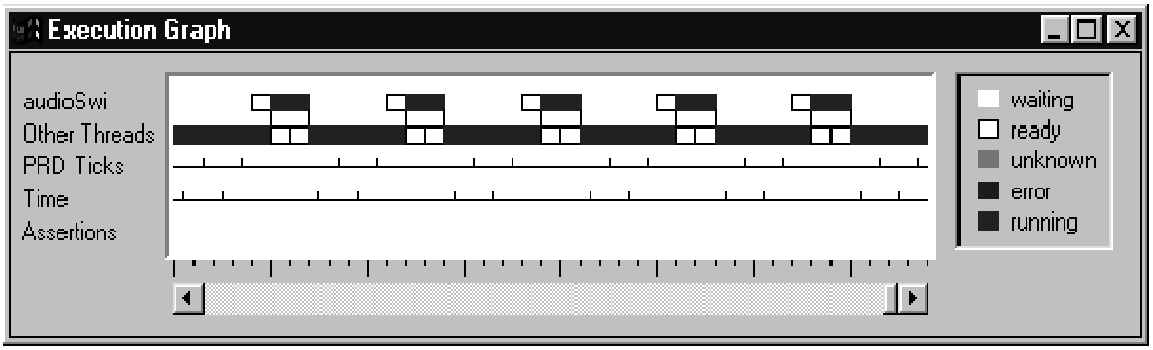


RTA Control Panel

☑ enable SWI logging
☐ enable PRD logging
☑ enable CLK logging
☑ enable SWI accumulators
☐ enable PRD accumulators
☐ enable PIP accumulators
☐ enable HWI accumulators
☐ enable USER0 trace
☐ enable USER1 trace

☑ global target enable
☑ global host enable

4. Right-click on the area that contains the check boxes and deselect Allow Docking, or select Float in Main Window, to display the RTA Control Panel in a separate window. Resize the window so that you can see all of the check boxes shown here.
5. Put check marks in the boxes shown here to enable SWI and CLK logging, SWI accumulators and globally enable tracing on the host.
6. Choose Tools –> DSP/BIOS –> CPU Load Graph.
7. Choose Tools –> DSP/BIOS –> Execution Graph. The Execution Graph appears at the bottom of the Code Composer Studio window. You may want to resize this area or display it as a separate window.
8. Right-click on the RTA Control Panel and choose Property Page from the pop-up menu.
9. Verify that the Refresh Rate for Message Log/Execution Graph is 1 second and click OK.
10. Choose Debug –> Run or click the (Run) toolbar button. You should be able to hear the acoustic signal out of the speaker. The Execution Graph should look similar to this:



11. Count the marks between times the **audioSWI** object was running. It should be running every 2 milliseconds. You can see that in between 2 consecutive execution of **audioSWI** there are 2 system clock ticks (each clock tick corresponds to 1 msec). The **audio** function

runs every timer there is a full frame in **DSS_rxPipe** and an empty frame in **DSS_txPipe**. The ISR is running every 1/48000 microseconds, writing and reading a word at a time. For 96 word frames, the ISR needs to execute 96 times before filling up a frame. Therefore, the **audio** should be running every 96/48000 microseconds, i.e., 2 milliseconds.

3.5 Adding a Periodic Object

We are going to add a new function to our application that runs periodically at intervals of 8 milliseconds. The code for this function, called **load**, is in **audio.c**.

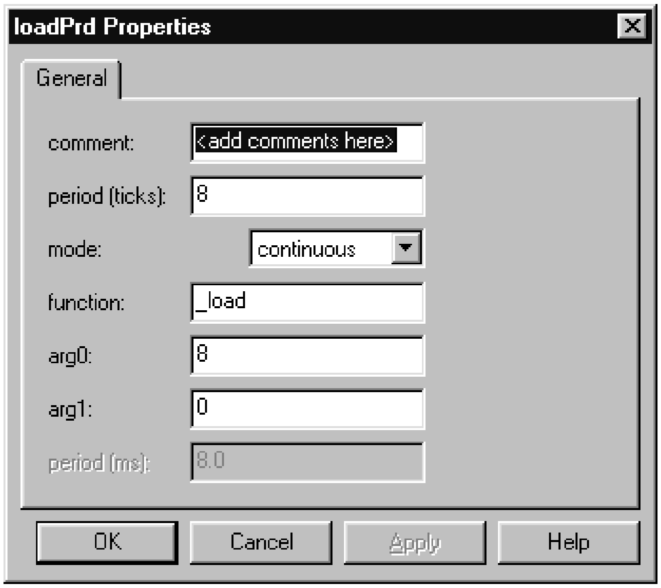
```
/*
 * ======== load ========
 */
Void load(Int prd_ms)
{
  static int oldLoad = 0;
  /* display confirmation of load changes */
  if (oldLoad != loadVal ) {
    oldLoad = loadVal;
    LOG_printf(&trace,
      ”load: new load = %d000 instructions every %d ms”, loadVal, prd_ms);
  }

  if (loadVal) {
    AUDIO_load(loadVal);
  }
}
```

loadVal is a global variable that is initially set to 0. AUDIO_load is an assembly routine that simulates a process that uses up cycles on the CPU. (You can find the source code for AUDIO_load in AUDIO_LD.S62.)

Using the DSP/BIOS Real-Time Analysis Tools you can set how much CPU load **load** takes up by changing the value of the global variable **loadVal**.

1. In Code Composer Studio, open **audio.cdb**. Right-click on the Periodic Function Manager and select Insert PRD from the menu.
2. Rename the new **PRD** object to **loadPrd**. Right-click on it and select Properties from the menu. Enter the properties for this periodic object as follows:



comment: <add comments here>
period (ticks): 8
mode: continuous
function: _load
arg0: 8
arg1: 0
period (ms): 8.0

Click ***OK*** to save your changes.

3. Choose Project –> Rebuild All and rebuild **audio.out**

3.6 Running With a Periodic Function

1. Choose File –> Reload Program menu item in Code Composer Studio to reload **audio.out**.
2. Choose Debug –> Run
3. Choose the View –> Watch Window menu item. In the Watch Window, Insert New Expression **loadVal**.
4. In the Trace State window, turn off **CLK** logging. Click on **PRD** logging.
5. Observe the Execution Graph window. You will see the **loadPRD** function executed every 8 system ticks. In the **STS** window for **audioSWI**, you will notice that the maximum value for the signal is well under the 2 millisecond deadline.
6. In the Data Memory window, double click **loadVal** and Edit Variable to **100**. Following this you should see the CPU load increase. The **AUDIO_load** function simulates a CPU load that is 1000 * **loadVal**. Do you observe this increase?
7. If you put too large of a value in **loadVal**, notice the CPU Load graph and the Execution Graph stop updating. This is because the updates are performed within an idle task, which

has the lowest execution priority within the program. Because the higher-priority threads are using all the processing time, there is not enough time for the host control updates to be performed. The program is now missing its real-time deadline. Putting a smaller value in **loadVal** will continue target/host communication.

8. Observe the **STS** data for **audioSWI**. The increase in the load of the periodic function raised the maximum, but it is still under the real time deadline of 2 milliseconds for **audioSWI**.
9. Increase **loadVal** by entering new values in the Data Memory window. Try **150**, **200**, etc. Observe the increase on the CPU load and the **Max** field for the **audioSWI STS**. Observe how eventually the **audioSWI STS** maximum reaches more than 2 milliseconds, and the application starts to miss real time. You will notice this as you start to hear the quality of the acoustic signal output getting worse. Write down the value of **loadVal** that causes the application to start missing real time.
10. In the System Log window, observe how the increase in **loadVal** lengthened the amount of time it took for **loadPRD** to execute. (You can see this by counting the number of system ticks that came in while **loadPRD** was running).

3.7 Increasing the Number of Frames

An increase in the load of **loadPRD** makes the example miss real time: when **loadPRD** takes too long to finish, it makes **audioSWI** start executing late, since **audioSWI** has to wait until **loadPRD** is done. With only 2 frames, the ISR may finish filling up (or transmitting) one of the frames before **audioSWI**, delayed by **loadPRD**, has had a chance to copy the other frame and release it back to the pipe. As a result, interrupts will occur before a new frame is available, resulting in a loss of data by the ISR. To address this problem you can increase the number of frames on each pipe, so there is another frame available for the ISR to fill up while **audioSWI** is delayed.

1. Open **audio.cdb** inside Code Composer Studio.
2. Open the Buffered Pipe Manager object, and highlight **DSS_rxPipe**. Right-click to bring up the Properties window and change the number of frames in the **numframes** field to **3**. Click ***OK*** to save your changes.
3. Repeat the same procedure with **DSS_txPipe**.
4. Choose Project –> Rebuild All and save the changes.
5. Reload **audio.out** and choose Debug –> Run.
6. Choose View –> Memory window, enter the value for **loadVal** that you had previously written down. Does the quality of sound deteriorate?
7. Observe the **STS** window for **audioSig**. Is the Maximum under 2 msec? Is **audioSWI** meeting its deadline?
8. Keep increasing **loadVal**. The application will eventually run into the same problem as before, and **audioSWI** will start missing real time again.

3.8 Getting Your Priorities Straight

Adding a new buffer seemed to solve the situation until **loadVal** became too large, and the application started to miss real time again. Why did this happen?

When **audioSWI** was posted, it did not run immediately but had to wait until **loadPRD** finished. When **loadPRD** started to take more than 2 milliseconds to execute, it caused the **audioSWI** waiting for it to be late for its 2 milliseconds deadline. When we added a new buffer, we increased the amount of time that **audioSWI** could be late, as there was an extra buffer that the ISR could continue to fill. However, this kind of arrangement will not help us indefinitely. If **loadVal** becomes large enough, eventually **audioSWI** will be delayed long enough that the ISR will run out of available frames.

To solve this situation we need **audioSWI** to be able to preempt the periodic software interrupt (**PRD_swi**) that calls **loadPRD**. If **audioSWI** can preempt **PRD_swi**, **audioSWI** will run whenever it is posted, regardless of whether **loadPRD** is finished, because it will take away the control of the CPU (preempt) from the periodic function. Since **audioSWI** is no longer delayed by **loadPRD**, the load of **loadPRD** will no longer make our application miss real time.

1. Open **audio.cdb** inside Code Composer Studio. Click on the Software Interrupt Manager object to highlight it.
2. On the right pane, drag **audioSWI** above **PRD_swi**. **audioSWI** becomes the highest priority software interrupt.
3. Choose Project –> Rebuild All and save the changes.

Follow the same steps as in the previous section and observe how **audioSWI** does not miss real time when **loadVal** is increased. Note in the Message Log window how **audioSWI** preempts **loadPRD**. Note in the **STS** window for **audioSWI** how the maximum remains constant regardless of changes to **loadVal**.

3.9 Reviewing the ISR Code: The Assembly Interface

The code for the serial port receive interrupt ISR can be found in **DSS_AISR.S62**. Most of the ISR is written in assembly. We will use the ISR code to illustrate the DSP/BIOS assembly interface.

```
    ;  ======== dss_aisr.s62 ========
    ;
            .include c62.h62
            .include hwi.h62
(1)         .include pip.h62

            .include dss.h62

    DRR             .set    0x018c0000      ; Data Receive Register McBSP 0
(2) DXR             .set    0x018c0004      ; Data Transmit Register McBSP 0
            .bss rtxDone,4,4                ; Allocate temp variable in .bss to
                                            ; allow loads via b14.  No cinit
                                            ; recordneeded because ISR writes to
                                            ; this location before it reads it.
            .text
            .global _DSS_isr, rtxDone, rxErr, txErr
    ;
    ; ====== _DSS_aisr ======
    ;
    _DSS_isr:
```

```
             stw a0,*b15--[2]                ; push temp registers
             stw a1,*b15--[2]
(3)          stw a2,*b15--[2]
             stw b1,*b15--[2]
             stw b2,*b15--[2]
             ;  rxDone = 0, txDone = 0
             zero a2
             ;    if (DSS_rxCnt) {
             ldw *+b14(_DSS_rxCnt),b1
             nop 4
      [!b1]  b rxErr                         ; process rx error
             ;    *DSS_rxPtr++ = *DRR;
      [b1]   mvkl DRR,a1                     ; load address of serial port DRR
      [b1]   mvkh DRR,a1
      [b1]   ldw *a1,a1                      ; read word from DRR
      ||[b1] ldw *+b14(_DSS_rxPtr),b1        ; load DSS_rxPtr
      [b1]   ldw *+b14(_DSS_rxCnt),b2        ; load DSS_rxCnt
             nop 3
(4)          stw a1,*b1++                    ; store DRR at *DSS_rxPtr, auto
                                             ; increment DSS_rxPtr
             stw b1,*+b14(_DSS_rxPtr)        ; store updated DSS_rxPtr
             ;       DSS_rxCnt--;
             sub b2,1,b2                     ; decrement DSS_rxCnt
             stw b2,*+b14(_DSS_rxCnt)        ; store updated DSS_rxCnt
             ;       if (DSS_rxCnt == 0) {
             ;           rxDone = 1;
             ;       }
             ;    }
       [!b2] mvk 1,a2
     checkTx:
             ;    if (DSS_txCnt) {

             ldw *+b14(_DSS_txCnt),b1
             nop 4
      [!b1]  b txErr                         ; process tx error
             ;    *DXR = *DSS_txPtr++;
      [b1]   ldw *+b14(_DSS_txPtr),b1        ; load DSS_txPtr
      [b1]   ldw *+b14(_DSS_txCnt),b2        ; load DSS_txCnt
             nop 3
             ldw *b1++,a0                    ; load word pointed to by DSS_txPtr
                                             ; autoincrement DSS_txPtr
(5)          stw b1,*+b14(_DSS_txPtr)        ; store updated DSS_txPtr
             mvkl DXR,a1                     ; load address of serial port DXR
             mvkh DXR,a1
             ;    DSS_txCnt--;
             sub b2,1,b2                     ; decrement DSS_txCnt
             stw a0,*a1                      ; write word to DXR
             stw b2,*+b14(_DSS_txCnt)        ; store updated DSS_txCnt
             ;       if (DSS_txCnt == 0) {
             ;           txDone = 1;
             ;       }
             ;    }
       [!b2] or 2,a2,a2
```

```
     checkDn:
                                             ;if ((rxDone | txDone) == 0) {
      [a2]   b Done                       ; if rxDone or txDone do Done processing
             stw a2,*+b14(rtxDone)        ; store done flags into memory
             ;       return;              /* return from interrupt */
             ;    }
(6)
      [!a2]  ldw *++b15[2],b2                ; restore temp registers
      [!a2]  ldw *++b15[2],b1
      [!a2]  ldw *++b15[2],a2
      [!a2]  ldw *++b15[2],a1
             b irp                           ; return from interrupt
             ldw *++b15[2],a0
             nop 4
     Done:
             ldw *++b15[2],b2                ; restore temp registers
             ldw *++b15[2],b1
             ldw *++b15[2],a2
             ldw *++b15[2],a1
             ldw *++b15[2],a0
             nop 4
             HWI_enter C62_ABTEMPS, 0, 0xffff, 0
             ;    if (rxDone) {
             ldw *+b14(rtxDone),b0           ; load done flags from memory
(7)          nop 4
             and b0,1,b0                     ; check if rxDone set
      [!b0]  b txDone
             nop 3
             ;     PIP_put(&DSS_rxPipe);
(8)   [b0]   mvkl _DSS_rxPipe,a4             ; load pipe address
      [b0]   mvkh _DSS_rxPipe,a4
             PIP_put
             ;     DSS_rxPrime();
             ;   }
             b _DSS_rxPrime
             mvkl txDone,b3                  ; set return pointer to come back here
             mvkh txDone,b3
             nop 3
     txDone:
             ;    if (txDone) {
             ldw *+b14(rtxDone),b0           ; load done flags from memory
             nop 4
             and b0,2,b0                     ; check if txDone set
      [!b0]  b allDone
             nop 3
             ;       PIP_free(&DSS_txPipe);
      [b0]   mvkl _DSS_txPipe,a4             ; load pipe address
(9)   [b0]   mvkh _DSS_txPipe,a4
             PIP_free
             ;       DSS_txPrime();
             ;    }
             b _DSS_txPrime
             mvkl allDone,b3                 ; set return pointer to come back here
```

```
        mvkh allDone,b3
        nop 3
(10) allDone:
        HWI_exit C62_ABTEMPS, 0, 0xffff, 0

rxErr:
        ;       dummy = *DRR;
        mvkl DRR,a1                       ; load address of serial port DRR
        mvkh DRR,a1
        ldw *a1,a1                        ; read word from DRR
  ||    ldw *+b14(_DSS_error),b1          ; load DSS_error value
        b checkTx                         ; start return to primary ISR code
        nop 3
        ;       DSS_error |= 1;
        or b1,1,b1                        ; DSS_error has now arrived
        stw b1,*+b14(_DSS_error)          ; save new value of DSS_error
txErr:
        ;       *DXR = 0;
        mvkl DXR,a1                       ; load address of serial port DXR
        mvkh DXR,a1
 ||     zero b1
        stw b1,*a1                        ; write to DXR
        ldw *+b14(_DSS_error),b1          ; load DSS_error value
        b checkDn                         ; start return to primary ISR code
        nop 3
        ;       DSS_error |= 2;
        or b1,2,b1                        ; DSS_error has now arrived
        stw b1,*+b14(_DSS_error)          ; save new value of DSS_error
        .end
```

1. Since the ISR calls **HWI** and **PIP** assembly macros, **hwi.h62**, **pip.h62** and **c62.h62** need to be included. The **HWI_Obj** and **PIP_Obj** structures and the **HWI** and **PIP** module macros are defined in these header files. These include files can be found in the **...\ti\c6000\bios\include** directory of your product distribution. The order in which these files are included is not important.
2. Set the addresses for the Multichannel Buffered Serial Port 0.
3. An ISR must save all the registers it uses. At the very beginning of **_DSS_isr** we save only a subset of the total registers used by the ISR. This is done to optimize performance and CPU consumption.
4. Every time the ISR is triggered, **checkRead** copies the 32 bit value from the Serial port Data

   Receive Register (**DRR**) to a word in a **_DSS_rxPipe** frame. **_DSS_rxCnt** (defined in **dss.c**) keeps track of how many words are left to fill up the current **_DSS_rxPipe** frame. **_DSS_rxPtr** (defined also in **dss.c**) points to the location in the current frame where the next data sample from the serial port will be written. Every time the contents of **DRR** are copied to the **_DSS_rxPipe** frame, **_DSS_rxCnt** is decreased by one and **_DSS_rxPtr** is increased by one, to point to the next location in the frame. When the frame is full and ready to be put in **_DSS_rxPipe**, _DSS_rxCnt becomes 0.

   If **_DSS_rxCnt** was 0 when the ISR is triggered, there is no frame available to write the received serial port data. This error condition is handled by **readerror**.

5. Every time the ISR is triggered, **checkWrite** copies a word from a **_DSS_txPipe** to the 32 bit Serial port DataTransmit Register (**DXR**). **_DSS_txCnt** (defined in **dss.c**) keeps track of how many words are left to be transmitted in the current frame. **_DSS_txPtr** (defined also in **dss.c**) points the location in the frame from where the next data sample will be copied to **DXR**. Every time a data sample is copied to **DXR**,

   _DSS_txCnt is decreased by one and _DSS_txPtr is increased by one, to point to the next location in the frame. When all the data in the frame has been transmitted, and the frame is ready to be recycled back to _DSS_txPipe, _DSS_txCnt becomes 0.

   If **_DSS_txCnt** was 0 when the ISR is triggered, there is no full frame available to write data to the serial port data transmit register. This error condition is handled by **writeError**.

6. Check whether the last received data sample filled up a **_DSS_rxPipe** frame or whether the last transmitted data sample emptied a **_DSS_txPipe**. If neither of these conditions is met, we restore the registers and return from the ISR.

7. If a **_DSS_rxPipe** frame is full, the ISR needs to call the **PIP_put** assembly macro to put the frame back to the pipe. Calling **PIP_put** may result in the **audio** function being posted. The ISR will also call the C function **_DSS_rxPrime** to allocate the next empty frame from **_DSS_rxPipe** (if available). In order to do this the ISR needs to:

- Save any registers that may be used by DSP/BIOS internally (since kernel macros will be called).
- Save any registers that may be used by the compiler for the C function.
- Disable the scheduler, so that if a software signal is triggered as a result of a call to a kernel macro, the signal does not start executing in the context of the ISR. Instead, the ISR should finish first and then re-enable the scheduler so that the software signal can be executed.

  To meet these requirements the ISR calls **HWI_enter C62_ABTEMPS, 0, 0xffff, 0**. **HWI_enter** is an assembly macro that disables the scheduler and saves all the registers specified by the masks

8. We need to meet the preconditions before calling **PIP_put**: **a4** should contain the address of the pipe object (**_DSS_rxPipe**). There are no postconditions for **PIP_put**.

9. As with **PIP_put**, we need to meet **PIP_free** preconditions by loading the pipe object address into **a4**. **PIP_free** does not have any postconditions.

10. At the end of the ISR **HWI_exit** is called to restore registers, re-enable the scheduler, and exit the ISR

3.10 Using a C ISR

1. In the **...\ti\c6000\examples\bios\audio** directory you will find a C version of the ISR in the file **dss_cisr.c**. To use the C version of the ISR, remove the DSS_AISR.S62 file from the project and add the DSS_CISR.C and DSS_ASM.S62 files.

2. Save the changes and rebuild **audio.out.**

3. Load **audio.out**. Run the application. Note how using the C ISR increases the CPU load over the assembly version of the ISR.

To use **DSS_cisr** as our ISR we need to write a small stub function in assembly that calls this **DSS_cisr**. This assembly function is named **_DSS_isr**, as this is the function name entered in the **HWI_INT11** object (therefore, the function that is plugged in the TMS320C62x interrupt vector table). The code for this function can be found in **DSS_ASM.S62**:

```
; ======== dss_asm.s62 ========
;
;
    .include c62.h62
    .include hwi.h62
    .global _DSS_isr
    .global _DSS_cisr
     .text
;
; ======== _DSS_isr ========
;
; Calls the C ISR code
;
_DSS_isr:
    HWI_enter C62_ABTEMPS, 0, 0xffff, 0
    b _DSS_cisr
    mvkl dssi,b3
    mvkh dssi,b3
    nop 3
dssi:
    HWI_exit C62_ABTEMPS, 0, 0xffff, 0
    .end
```

_DSS_isr needs to save all the registers that it uses. Therefore, it needs to use the HWI_enter/HWI_exit macros to preserve all those registers that are not saved by the compiler. It also needs to make sure that the scheduler is disabled, as DSS_cisr will call DSP/BIOS APIs that may post a software interrupt.

3.11 Things to Try

- Increase the size of the frames in the pipes. Observe the System Log and note any changes on the frequency at which the signal runs. What happens when we double the frame size?
- Change the frame size in the pipes to 512 words. Observe the changes in the CPU load. Observe the System Log. When you increase the buffer size to 8 times the original value, the **PIP** operations (**PIP_alloc**, **PIP_put**, **PIP_get**, **PIP_free**), and the posting of the **audio** function occur at a frequency that is 8 times smaller. (You can check this by looking at the System Log). However, this does not cause an equivalent reduction on the CPU load. This is an indication that the **PIP** operations and the software interrupt scheduler have a small overhead in the overall CPU load for this application. Servicing the ISR and copying the data remain the largest overhead on the CPU.
- Enable the monitors on the signal side of the pipes. Observe the frequency at which frames flow in and out of the pipes, and how it changes with frame size. Note the impact of the monitors on the CPU.
- Enable a monitor for **HWI_INT11** and observe its impact on the CPU. In general, you can observe how enabling instrumentation (logging, accumulators, monitors) in the application impacts the CPU load. Notice if there is a dependency with the frame size of the pipes.

4 Summary/Conclusion

This audio example has demonstrated how to configure and use DSP/BIOS APIs for scheduling data transfer between the hardware I/O peripherals and the target DSP. This example has been provided to assist with your DSP application development and the understanding DSP/BIOS.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 1999, Texas Instruments Incorporated