Application Report
SPRA653 - April 2000

Understanding Basic DSP/BIOS Features

Henry Yiu *Texas Instruments, China*

ABSTRACT

DSP/BIOS is a small firmware that runs on digital signal processor (DSP) chips. It provides the software components that allow application developers to have the following two basic capabilities:

- Real-time monitoring and control – monitor and control program execution and program variables in real time
- Real-time scheduling – real-time multi-threaded system scheduling and communication

Texas Instruments *TMS320C6000 DSP/BIOS User's Guide* (Literature Number SPRU303) provides a detailed description of the DSP/BIOS features and the application program interface. This application report is intended to give the user a short overview of the basic TMS320C6000™ DSP/BIOS features so that one can have some idea on how to implement a complete DSP system using DSP/BIOS.

Contents

List of Figures

List of Tables

1 What is DSP/BIOS?

For most system developers today, more time is spent on software design than on hardware design. This is mainly due to the fact that most of the complex hardware functions are already integrated into standard integrated circuits. System developers simply need to select the right chip to do the right job.

System designers can use either dedicated hardware integrated circuits or programmable DSPs to implement DSP functions. The advantages of programmable DSPs are:

- Scalability – Designers can first decide how much processing power (or MIPS—million instructions per second), memories, and other resources are required to do a certain job, then select the DSP with enough processing resources. If one DSP is not enough, more DSPs can be added until the required processing resources are met.

- Upgradability – Even after the hardware design is fixed, any change in DSP functions can be implemented easily by making changes in software, and downloading to the DSP in seconds.

However, compared to dedicated hardware DSP chips to implement DSP functions, programmable DSPs need to manage their resources so that they can be shared among different DSP functions. Hardware resources include the CPU, memory, and peripherals.

- CPU-sharing – In most system, the number of DSP functions is much larger than the available number of CPUs. Thus, each CPU must be time-shared among DSP functions. But time-sharing the CPU may cause a critical DSP function to miss its real-time deadline. DSP/BIOS and Code Composer Studio™ are the tools to make time-sharing CPU much easier and avoid real-time deadline problems.

- Memory-sharing – Memory-sharing is usually the responsibility of the algorithm writer. Using the C language, static memories can be managed easily. However, when the system gets complex with lots of DSP functions, it is not efficient to store each algorithm's scratch memory statically. Texas Instruments eXpressDSP™ Algorithm Standard (xDAIS) addresses this issue. xDAIS is out of the scope of this application report.

- Peripheral-sharing – In most cases, such as I/O devices, a peripheral can be handled by a dedicated DSP function, which is called a device driver. Peripherals such as the direct memory access controller (DMAC) can also be handled this way.

With DSP/BIOS, system designers can develop their DSP systems on programmable DSPs using block diagrams to block out each DSP function, similar to designing using dedicated DSP hardware integrated circuits.

As mentioned earlier, DSP/BIOS allows system designers to do two major tasks: real-time monitoring and control, and real-time scheduling and communication. DSP/BIOS provides modules to accomplish these tasks (see Table 1).

Table 1. DSP/BIOS Modules

Modules for system configuration	
GBL	Global setting manager
MEM	Memory manager
Modules for real-time monitoring and control	
LOG	Message log manager
STS	Statistic accumulator manager
TRC	Trace manager
Modules for real-time scheduling	
HWI	Hardware interrupt manager
SWI	Software interrupt manager
IDL	Idle function and processing loop manager
CLK	System clock manager
PRD	Periodic function manager
Modules for real-time communication	
PIP	Data pipe manager
HST	Host input/output manager
RTDX	Real-time data exchange manager

The modules for system configuration (GBL and MEM) define the hardware and system environments. These settings should match the actual target hardware environment.

The modules for real-time monitoring and control (LOG, STS, and TRC) provide a mean to send information to the host while the target program executes. LOG is used to send messages, such as using a LOG_printf statement. STS is used to gather statistical information of a value, such as average, maximum, and count of occurrence of the value. TRC controls the on and off of information capture. It also provides two bits (TRC_USER0 and TRC_USER1) to allow the host to control the flow of the target program in real time.

The modules for real-time scheduling contain three basic items called threads: HWI, SWI, and IDL. These objects contain routines to handle a particular task in a real-time environment. These routines could be the DSP functions that the system designer would like to implement on the programmable DSP. HWI sets the interrupt service routines for the interrupt vector table and provides some functions to manage hardware interrupts. The HWI object is the hardware interrupt service routine triggered by hardware. SWI manages software interrupt service routines with SWI objects. The SWI service routines are triggered by a call to the SWI_post() function. IDL manages the background tasks, and it is triggered whenever no SWI and HWI services routines are pending. HWI objects are highest-priority threads. SWI objects are next in the priority. IDL objects are lowest in priority.

CLK objects run in the context of the timer hardware interrupt (HWI_INT14) service routine. PRD objects are a special type of SWI, running in the context of the period software interrupt (PRD_swi) service routine. The PRD_swi object can be selected so that it is triggered by one of the CLK object (PRD_clock) or by a call to the PRD_tick() API function.

The modules for real-time communication (PIP, HST, and RTDX) manage communication channels between threads, and between the target and the host. The PIP and HST objects are single contiguous buffers similar to hardware FIFOs. For communication synchronization purpose, these buffers can be set so that when the buffer has just been written or has just been read, any function is called automatically. For instance, when the buffer is read, a call to SWI_post() can be used to trigger a SWI interrupt service routine to put data into the buffer. This triggering method is called to notify the reader or the writer.

2 Software Design Using Block Diagrams

An example implementation of DSP/BIOS is the AUDIO DEMO example supplied with the Code Composer Studio. A system designer can design a DSP system by first knowing what kind of DSP functional blocks he would like to have. These DSP functional blocks would be placed in the SWI, HWI, or IDL thread objects, depending on their triggering methods and their priorities. If the DSP function requires a periodic trigger, it can be placed in the PRD or CLK thread objects. Data transfer between threads can be drawn using thick black lines via PIP objects. Then, after the functional blocks are drawn, the system designer should decide how these functional blocks are triggered. For example, triggering the audio() function requires both a filled input buffer and an empty output buffer. The triggering signals can be drawn via dotted lines.

depicts the block diagram of the AUDIO DEMO example. By default, the DSP/BIOS configuration tools have already created the blocks in dotted lines. The name of the module type is placed at the upper left corner of each block (such as SWI). The name of the module object is placed inside each block (such as audioSwi). If the block is a thread object, the corresponding function is placed under the name of the object (such as audio()). Then, the flow of data is connected via communication objects (such as PIP). Decide how to trigger the thread objects. HWI must be triggered from a hardware signal. In the example, HWI_INT11 is triggered by the receiver data available signal from the serial port hardware interrupt. Since the transmit and receive data rates are the same, data available from the receiver register DRR can also be sent out via the transmit register DXR. For SWI, the trigger can be directly from another SWI or HWI that calls a SWI_post function, or from a PIP notify function to indicate that the data channel is ready. In the example, the audioSwi is triggered by a write to the DSS_rxPipe and a read from the DSS_txPipe. The AND function is implemented using the SWI_andn() API function, which is a derivative of the SWI_post() function.

Figure 1 makes the relationship between threads easier to understand. This understanding will be useful in troubleshooting any real-time problems that may occur once the code is written.

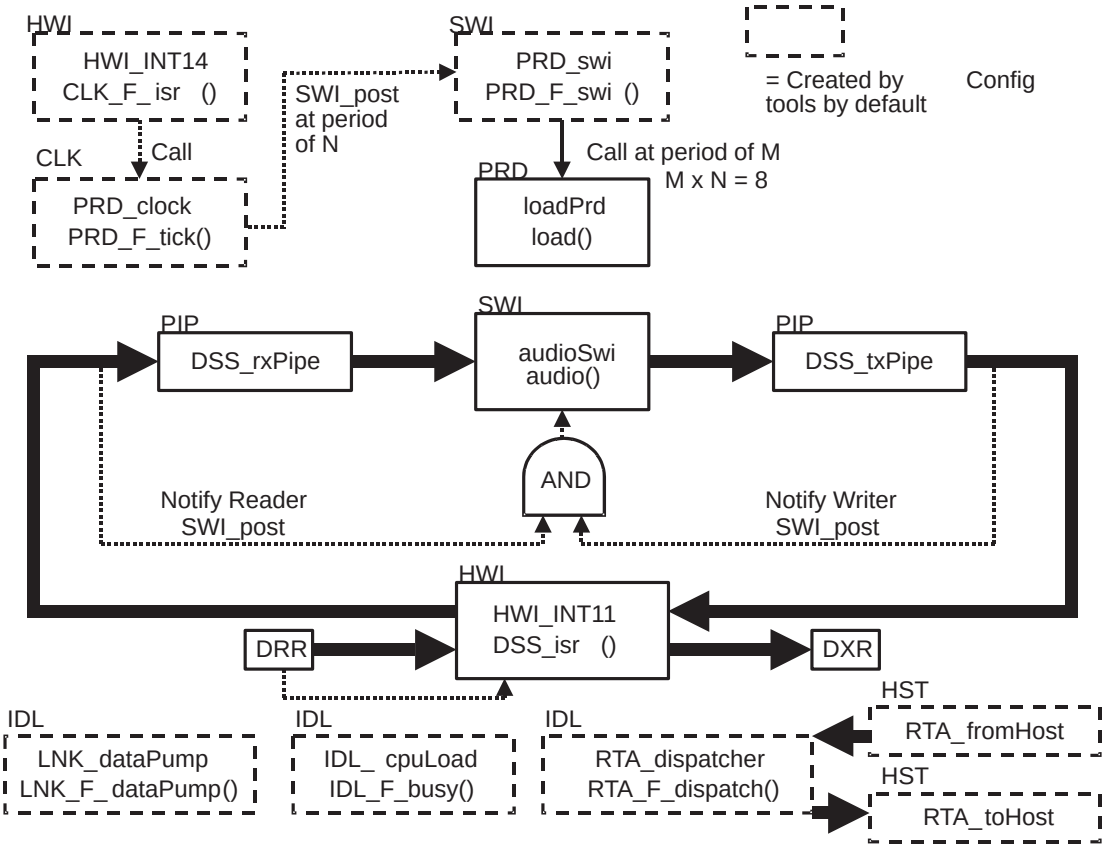


Figure 1. Block Diagram of the AUDIO DEMO Example

Figure 2, Figure 3, and Figure 4 depict the inner operation of the function of each thread object. The system designer can usually start writing code without drawing these diagrams. It is shown here to improve readability and to make the code easier to understand. The audio() function uses two variables to control its operation: noise_sw and filter_sw. When the Code Composer GEL modifies these variables, it is not done in real time since the target program must be halted before variables can be modified. To control program operation in real time, one needs to use the TRC modules.

Note that we use a DSS_isr assembly routine to call the C interrupt service routine. This is necessary so that we can call the HWI_enter and HWI_exit assembly macros. These macros disable software interrupts, thus, ensuring that the audioSwi is run only after the HWI has exited.

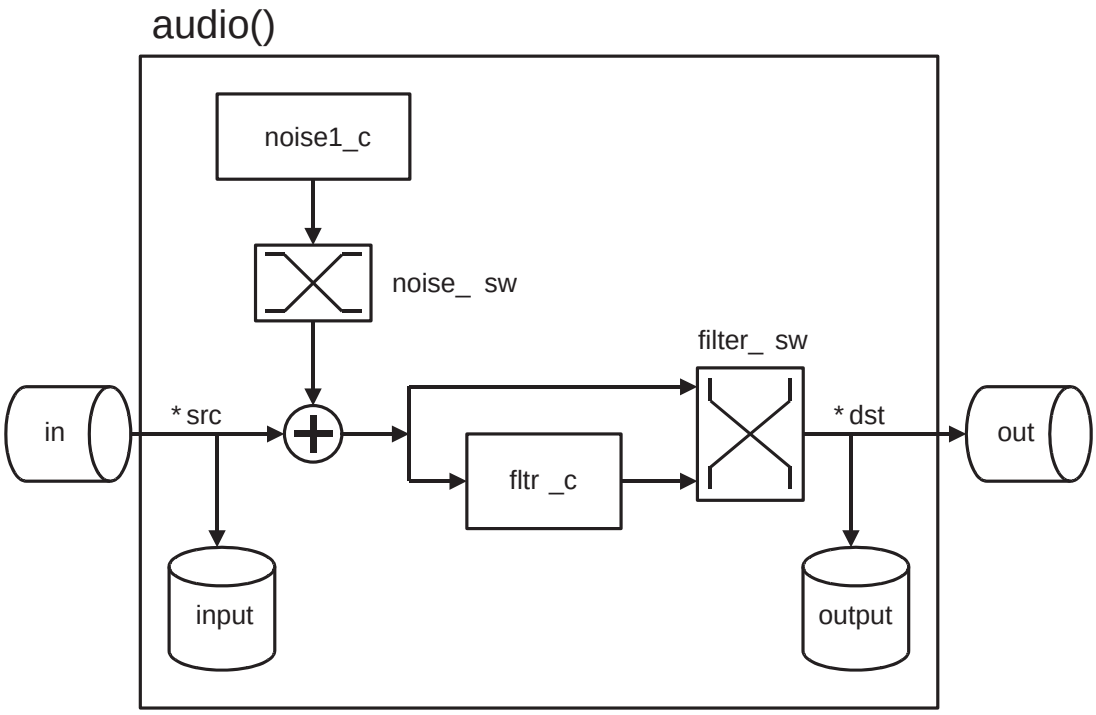
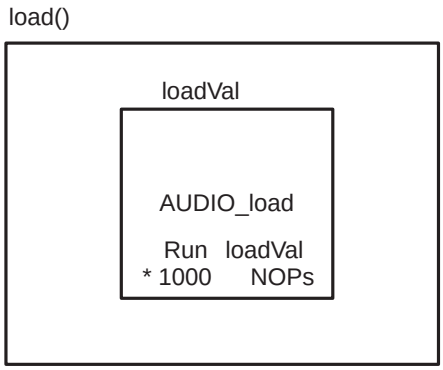


Figure 2. Block Diagram and Code of the audio() DSP Function in the AUDIO DEMO Example

```
Void audio(PIP_Obj *in, PIP_Obj *out)
{
    Int  *src, *dst;
    Int  size, indx;

    while (PIP_getReaderNumFrames(in) && PIP_getWriterNumFrames(out)) {

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

        PIP_alloc(out);
        /* get data buffer pointer and frame size */
        src = PIP_getReaderAddr(in);
        dst = PIP_getWriterAddr(out);
        size = PIP_getReaderSize(in);
        PIP_setWriterSize(out,size);

        /* copy into a input buffer for Graph Display */
        for(indx=0; indx<96; indx++)
        {
            input[indx] = *(src+indx);
        }

        /* noise switch */
        switch (noise_sw)
        {
            case 0:
            break;
            case 1:
                noise1_c (src, size);
            break;
            default:
            break;
        }

        /* filter switch */
        switch (filter_sw)
        {
            case 0:
                cpy(dst, src, size);
            break;
            case 1:
                fltr_c (dst, src, size);
            break;
            default:
                cpy(dst, src, size);
            break;
        }

        /* copy into a output buffer for Graph Display */
        for(indx=0; indx<96; indx++)
        {
            output[indx] = *(dst+indx);
        }
        /* output copied data and free input buffer */
        PIP_put(out);
        PIP_free(in);
    }
}
```

Figure 2. Block Diagram and Code of the audio() DSP Function in the AUDIO DEMO Example (Continued)




```
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);
    }
}
N       .set    1000
_AUDIO_load:
        mv    a4, b0            ; use b0 as loop counter
        mvk   N, b1
        mpy   b1, b0, b0
        nop
        shru  b0, 3, b0         ; (loop counter)= (# loops)/8
loop:
        sub   b0, 1, b0
        nop
   [b0] b     loop
        nop   5
        b     b3
        nop   5                 ; return
```

Figure 3. Block Diagram and Code of the audio() DSP Function in the AUDIO DEMO Example

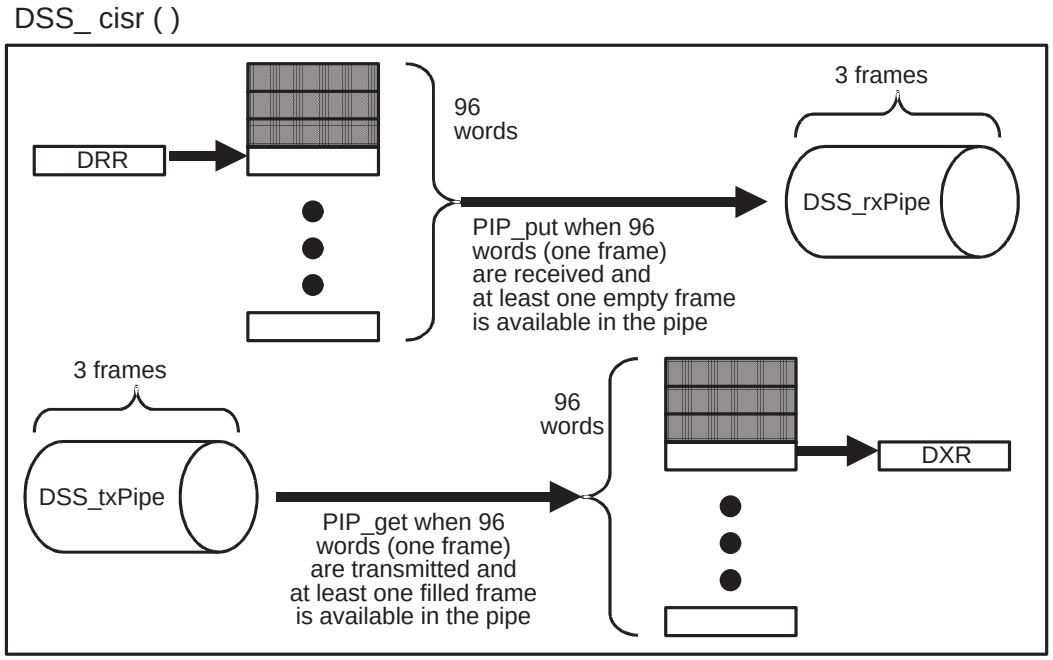


Figure 4. Block Diagram and Code of the load() DSP Function in the AUDIO DEMO Example

```
void DSS_cisr(void)
{
    volatile int dummy;
    if (DSS_rxCnt == 0 && PIP_getWriterNumFrames(&DSS_rxPipe) > 0) {
        PIP_alloc(&DSS_rxPipe);
        DSS_rxPtr = PIP_getWriterAddr(&DSS_rxPipe);
        DSS_rxCnt = PIP_getWriterSize(&DSS_rxPipe);
    }
    if (DSS_rxCnt) {
        *DSS_rxPtr++ = DRR;
        DSS_rxCnt--;
        if (DSS_rxCnt == 0) {
            PIP_put(&DSS_rxPipe);
        }
    }
    else {
        dummy = DRR;
        DSS_error |= 0x1;
    }
    if (DSS_txCnt == 0 && PIP_getReaderNumFrames(&DSS_txPipe) > 0) {
        PIP_get(&DSS_txPipe);
        DSS_txPtr = PIP_getReaderAddr(&DSS_txPipe);
        DSS_txCnt = PIP_getReaderSize(&DSS_txPipe);
    }
    if (DSS_txCnt) {
        DXR = *DSS_txPtr++;
        DSS_txCnt--;
        if (DSS_txCnt == 0) {
            PIP_free(&DSS_txPipe);
        }
    }
    else {
        DXR = 0;
        DSS_error |= 0x2;
    }
}
_DSS_isr:
        HWI_enter C62_ABTEMPS, 0, 0xffff, 0
        b    _DSS_cisr
        mvk  diss,b3
        mvkh diss,b3
        nop  3
diss:
        HWI_exit C62_ABTEMPS, 0, 0xffff, 0
```

Figure 4. Block Diagram and Code of the load() DSP Functionin the AUDIO DEMO Example (Continued)

3 Checking for Real-Time Problems

After the program is built, you can use the Execution Graph in Code Composer (see Figure 5) to see a visual display of thread activity, or use the Statistics View to see the statistics information on the execution time of each thread. In order to use the Execution Graph, the thread “logging” from the RTA Control Panel must be enabled. In order to use the Statistic View to gather statistics information on the execution time, the thread “accumulators” from the RTA Control Panel must be enabled. The statistics gathered for SWI functions measure the time from when a software interrupt is ready to run, to the time when it completes. If the maximum of this ready-to-complete time is close to or even longer than the allowed period of the SWI, then it is likely that the system will have real-time problems. It is important to note that these real-time analysis tools only provide real-time analysis information to the user. It is up to the user to determine whether the program meets real time or not. Thus, the definition of meeting real-time depends on application.

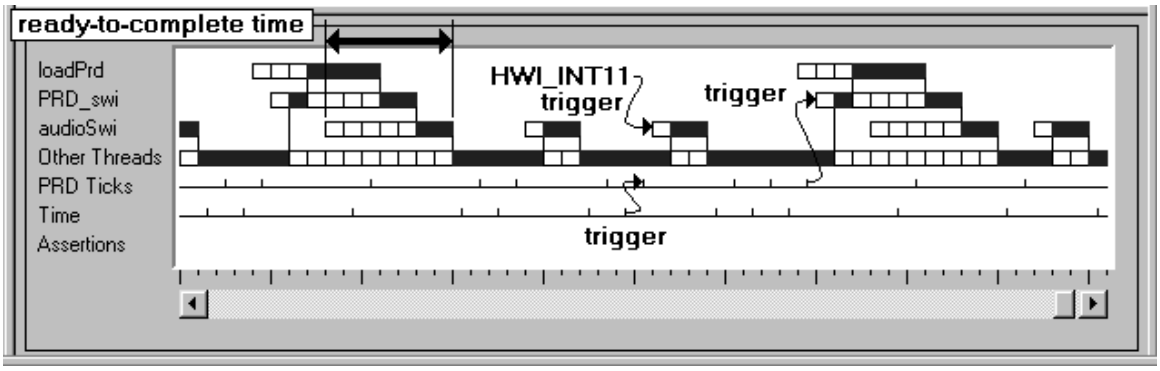


Figure 5. Execution Graph of the AUDIO DEMO Example

In the AUDIO DEMO example, we know that the input and output audio data is coming from and going to an audio codec on the TMS320C6201 EVM. It is set up to run at 48-kHz stereo 16-bit format. Each sample of the stereo data is packed into a 32-bit word. The DSS_cisr() function collects 96 words before it is passed to the pipe as one frame. Since the audio() function processes one frame at a time, the nominal frequency of the audio() function should be 48 kHz / 96 = 500 Hz. Therefore, if the maximum ready-to-complete time of the audioSwi object does not exceed the period of 1 / 500 Hz = 2 ms, then it is most likely that there should not be a real-time problem. By adjusting the loadVal variable, we can see the effect of loadVal to the maximum ready-to-complete time of the audioSwi object. As seen in Table 2, when audioSwi has a lower priority than PRD_swi and loadVal exceeds about 100, then the maximum ready-to-complete time is over 2 ms.

Table 2. AUDIO DEMO Maximum audioSwi Ready-to-Complete Time Versus Loading

loadVal	audioSwi higher priority than PRD_swi	audioSwi lower priority than PRD_swi
0	0.02 ms	0.03 ms
100	0.02 ms	0.89 ms
300	0.02 ms	2.62 ms
500	0.02 ms	3.34 ms

Although the audioSwi processes one frame of data at a time, it does not mean that its period has to be less than the frame period to avoid real-time problem. For example, one can increase the length of the PIP objects so that the audio() function can wait for more frames to be available before it processes several of them at once. This is possible since the frequency of the PRD_swi is at 125 Hz, which is four times less than the data frame rate of 500 Hz. Therefore, as before, the definition of what is real time depends on the application.

In the case of the AUDIO DEMO example, we can define real time to be: all data received from the DRR is processed and sent back to the DXR, with no loss of data. If this is the definition, we can check it by checking the DSS_error variable to see if it is non-zero. This variable is set in the DSS_cisr() function when data received in the DRR cannot be placed into the receive pipe because it is still full, or when no data is available at the transmit pipe when the DXR requests data. This error occurs when the audio() function cannot process data fast enough.

One cannot halt the DSP in order to check the DSS_error variable. This is because the audio codec hardware is still interrupting the DSP while the DSP is halted. A way to check the DSS_error variable is to use a TRC module. The DSS_error is always set to zero until the user click on the “USER0 trace” box in the RTA Control Panel. This My_idle() function can be set as an IDL object to check for real-time problem.

```
Void My_idle()
{
    if (TRC_query(TRC_USER0))
        DSS_error = 0;
    if (DSS_error & 0x1)
        LOG_printf (&trace, ”DRR receive error”);
    if (DSS_error & 0x2)
        LOG_printf (&trace, ”DXR transmit error”);
}
```

To check for real-time problem, first uncheck the “USER0 trace” box, then run the AUDIO DEMO program. Wait a few seconds until the flow of data is normal, and then check the “USER0 trace” box. If no error is seen from the “trace” log after it has been running for a while (maybe a few minutes), then the chance of a real-time problem is quite small. Since changing loadVal via Code Composer GEL is not a real-time process, the “USER0 trace” box must be disabled anytime loadVal is changed.

Table 3. AUDIO DEMO Occurrence of Real-Time Problem Versus Loading, Where audioSwi is Lower Priority Than PRD_swi

Sizes of DSS_txPipe and DSS_rxPipe (frames)	Pre-filled frames in DSS_txPipe (frames)	Maximum loadVal before real-time problem occurred	CPU Loading (%)
2	0	255	41.6
2	1	255	42.2
3	1	555	67.4
4	1	595	70.8
4	2	595	71.0
5	2	930	98.5
6	2	930	98.5
6	3	930	98.5

To use this method to check for real-time problems, refer to Table 3, which shows the effect of the sizes of the transmit and receive pipes on the maximum loading before real-time problems occur. Note that when the size of the pipes increases, the transmit pipe has to be pre-filled to a certain degree. This allows the audioSwi to be slower than the HWI_INT11. Pre-filling can be done easily in the main() routine by using the PIP_alloc() and PIP_put() API functions. The following shows how to pre-fill one frame for the DSS_txPipe.

```
Void main()
{
    DSS_init();
    LOG_printf(&trace, ”Audio example started!!\n”);
    PIP_alloc(&DSS_txPipe);    /* Pre-fill one frame */
    PIP_put(&DSS_txPipe);
    /* fall into BIOS idle loop */
    return;
}
```

It is proved here that for the AUDIO DEMO example, increasing pipe sizes reduces the chance of real-time problem under loading.

4 Conclusion

This report provided a short overview of the basic features of the DSP/BIOS real-time kernel. A system designer who wishes to develop DSP functions on a programmable DSP must be very knowledgeable about each of the DSP functional block, its processing power requirement, what resources are required, and how it is triggered to run. With the aid of the DSP/BIOS and Code Composer real-time analysis tools, the system designer can check for any real-time problems while the functional blocks are constructed. This will avoid any real-time glitches, which are very difficult to fix once the system is fully built.

5 References

1. TMS320C6000 DSP/BIOS User’s Guide, TI Literature Number SPRU303, Texas Instruments, May 1999.

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 acknowledgment, 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.

Customers are responsible for their applications using TI components.

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.