Texas Instruments

Application Report
SPRA695 - October 2000

Real-Time DSP Software Design for a Portable MP3 Player on a Texas Instruments TMS320C54x DSP Using DSP/BIOS

Mathew George, Jr. (Joe)
Jack Greenbaum
Stephen Handley
Jason Kridner

Digital Signal Processing Solutions

ABSTRACT

As real-time digital signal processing (DSP) systems become more complicated with multiple threads of execution and faster DSPs, software development and debugging becomes more complicated. An MP3 player is a good example of a multi-threaded system that executes not only DSP-type (data) threads such as MP3 decoding, sample rate conversion, and graphic equalization, but it also runs typical "host/controller" (control) threads such as user interface (keypad/LCD) and access to storage media (for example, Compact Flash). In this document we discuss the software behind a real-world MP3 player running stand-alone on a Texas Instruments C5000™ DSP. Parts of this system are presently being shipped in production MP3 players all over the world. The software is built upon the Texas Instruments DSP/BIOS offering with scheduler and real-time analysis. This application report shows a step-by-step real-world software paradigm shift. We will start with a typical ISR (interrupt service routine) system and build it up in steps to incorporate such constructs as a real-time preemptive kernel and data piping structure. Then we will see how DSP/BIOS real-time analysis (RTA) can be used to observe and debug the entire system in real time. These debug features include scheduling graphs, loading calculations, and statistical benchmarking information.

Contents

1 Introduction **2**

2 Software Architecture Module Overview **2**
2.1 DSP Subsystem 3
2.2 Controller Subsystem 4

3 Typical Implementation (DSP Subsystem Only) **4**

4 Adding DSP/BIOS to the MP3 Player (DSP Subsystem) **6**
4.1 HWI/(MEM) Objects (First degree) 6
4.2 SWI Objects (Third Degree) 9
4.3 PIP Objects 12
4.4 Adding the Controller Subsystem (and the PRD object) to the MP3 Player 15

C5000 is a trademark of Texas Instruments.

5 DSP/BIOS RTA (Second Degree – with Implicit Instrumentation Objects) 17
5.1 Execution Graph and TRC/LOG//HST/RTDX/IDL Objects 17
5.2 Load Graph and the CLK Object 19
5.3 Real-time Deadlines, Benchmarking, and the STS Object 19
5.4 MP3 Player System Real-time Thread Summary 21
6 Other Benefits 23
7 Conclusion 25

List of Figures

Figure 1. System Module Overview (static) 3
Figure 2. Typical Implementation (DSP Subsystem Only) 4
Figure 3. DSP/BIOS Objects Summary 6
Figure 4. Configuration Tool Setup for HWI (/MEM) 7
Figure 5. Adding HWI Objects to the DSP Subsystem 8
Figure 6. Configuration Tool Setup for SWI 10
Figure 7. Adding SWIs to the DSP Subsystem 11
Figure 8. Configuration Setup Tool for PIP 13
Figure 9. Adding PIPs to the DSP Subsystem 14
Figure 10. Controller Subsystem 15
Figure 11. Brief DSP/BIOS RTA (Execution Graph/Log and Load Graph) 18
Figure 12. Detailed DSP/BIOS RTA with STS Accumulator 20
Figure 13. Simple Real-Block Diagram Summary of MP3 Player Threads 21
Figure 14. Power Management in DSP/BIOS 24

List of Tables

Table 1. Real-time Summary of MP3 Player Threads 22

1 Introduction

This application report introduces the concepts of real-time software design using DSP/BIOS of a portable MP3 player on a TI TMS320C54x DSP. The steps are discussed in stages, or degrees, to follow the concepts in *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (literature number SPRA591). This application report describes these degrees in a slightly different order, but should give a good, real-world, production SHIPPING example of DSP/BIOS. Much of the implementation described was executed on a TI Internet Audio Evaluation Module (IA EVM).

2 Software Architecture Module Overview

Figure 1 shows the static block diagram of how the functions of the MP3 player are divided into DSP and the Controller Subsystem functionality.

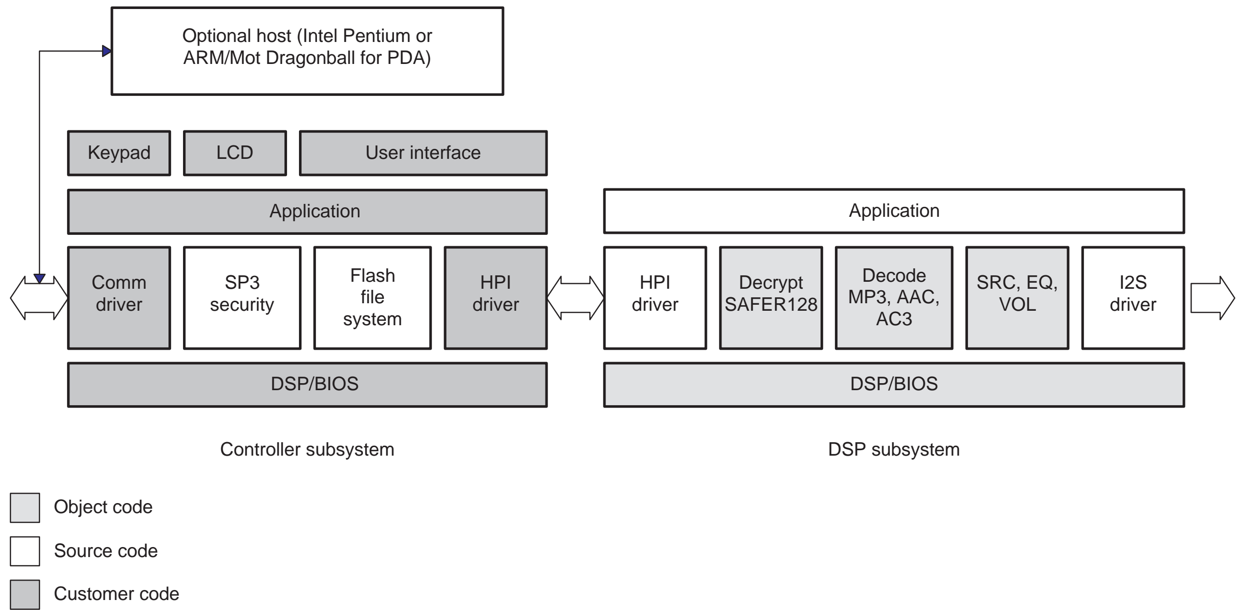


Figure 1. System Module Overview (static)

Though both control and data (Controller and DSP) subsystems are run on a single DSP, it is easier to visualize the two functions as separate. The DSP subsystem obeys player commands like STOP, PLAY, FFWD, etc. The Controller subsytem handles the user interface (keypad/LCD) and access to the Compact Flash storage media. We will reserve the term “Host” for a Pentium or PDA-type platform from which MP3 files may be transferred to the player after downloading from the Internet.

2.1 DSP Subsystem

Figure 1 shows the DSP subsystem code modules on the right side. The most important module is the Decode module in the middle block. Sample rate conversion (SRC – between 8 and 48 kHz), graphic equalization (EQ – like Rock, Pop, etc.) and volume control (VOL – about 64 steps) are included on the block to the right of the decoder. Any cryptology is supported in the Decrypt module to the left of the Decode block. All of this code is run using the DSP/BIOS real-time kernel. DSP/BIOS sequences the execution of the controller and DSP functions so that real-time deadlines are met. As we will see in section 5, DSP/BIOS also provides a detailed visibility into the execution of the application for debugging and optimization.

To the right of these modules are the codec drivers (I2S) that support I2S codec/DACs for audio playback. To the left are the Host Port Interface (HPI) drivers that support the command interface to the DSP subsystem over the DSP’s asynchronous slave port. The I2S drivers are controlling the C5000 Multichannel Buffered Serial Port (McBSP). This port can be configured to interface gluelessly to virtually any serial codec/DAC. The HPI drivers allow connection of the command interface to any master (or even slave) hardware interface. In this application report we will concentrate on a DSP-only build. In this case, these HPI drivers are written to allow the DSP to be its own master.

2.2 Controller Subsystem

The Controller subsystem modules are shown on the left side of Figure 1. This subsystem includes the Compact Flash file system (ATA-based) and Liquid Audio SPT security (if so licensed). The HPI driver includes the command interface to the DSP Subsystem. The keypad/LCD modules are specific to the board being used, in this case, the IA EVM. Since the IA EVM is stand-alone, the Comm Drivers are presently implemented using the TI CCS1.x tools over the JTAG scan-based emulation interface. The DSP/BIOS scheduler is used on the IA EVM as the “operating system” for the Controller subsystem.

Now we have a general understanding of how the code is structured. Consider how the system would be implemented to run in real time without DSP/BIOS. In sections 3 to 4.3 we consider how the system would be implemented to run in real time without DSP/BIOS for simplicity and academic purposes. Section 4.4 begins the discussion of implementing the controller subsystem using DSP/BIOS and how that solves implementation problems discovered when not using DSP/BIOS. Finally, section 5 shows how using DSP/BIOS provides further benefit to you and visibility into the real-time behavior of the system.

In sections 3 to 4.3 we build on the DSP subsystem example for simplicity and academic purposes. Section 4.4 to the end of the document discusses the Controller subsystem.

3 Typical Implementation (DSP Subsystem Only)

The following block diagram in Figure 2 shows a typical control/data flow implementation of the MP3 Player system components seen in Figure 1.

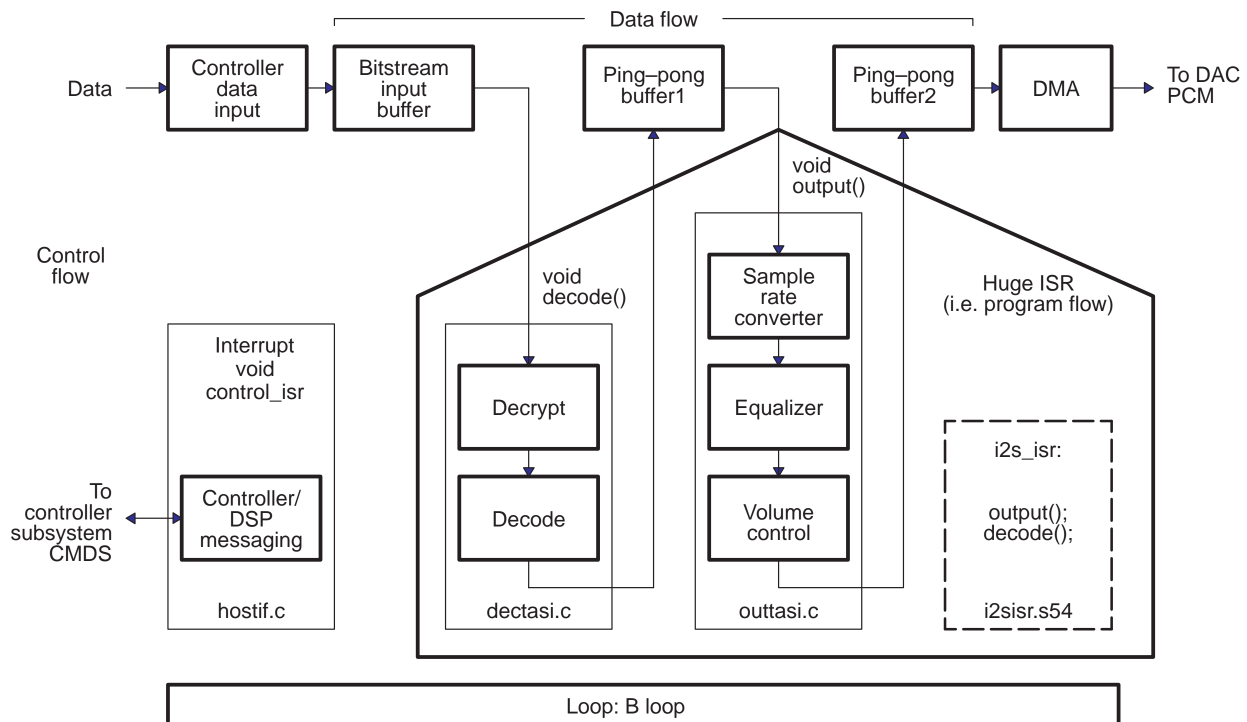


Figure 2. Typical Implementation (DSP Subsystem Only)

For simplicity in this documentation, only the DSP subsystem is shown. We will cover the Controller subsystem in Section 4.4. The MP3 Player code runs off the McBSP transmit interrupt (seen on the top right side of Figure 2) generated due to data required from the serial DAC in a “feed me” type architecture that cascades leftwards through the system/figure. Each 16-bit data packet is actually requested by the McBSP serial port transmit interrupt called XINT. XINT is running at the 44.1 kHZ CD-quality sample rate of the I2S audio codec/DAC.

The XINT is actually processesed by a DMA (Direct Memory Access controller), and when a requisite amount of samples are transmited (probably half the ping-pong buffer), then a DMA interrupt is sent to the CPU. Note that the DSP is probably sitting in a busy loop while waiting for the interrupt as seen at the bottom of Figure 2.

```
Loop: B loop
```

Then, inside a huge ISR based on the DMA interrupt, output and decode functions could be called that do all the requisite processing. Thus following from right to left in Figure 2, i2s_isr calls output() and then decode. Output() reads data from ping pong buffer-1 and first does SRC (sample rate conversion) since what ever sample rate the MP3 file was recorded at needs to be changed to 44.1 kHz to match the codec/DAC. It then implements a graphic equalizer (Rock, Pop, etc.) depending on user selection. Finally output() does a gain function for volume control and writes data to ping pong buffer-2. Following back to the decode() function call, the MP3 file data is read from the Bitstream Input Buffer, decrypted (if so licensed), decoded by whatever appropriate decoder, and then written to ping-pong buffer-1. The data is coming from a buffer (written by a host to the C54x HPI) before it cascades through the ping pong buffers left-to-right mentioned above as it makes its way to the DAC.

The above example is single threaded. A second thread on Figure 2 involves the Controller command interface to the DSP subsystem. It needs to know when to PLAY, STOP, FFWD, etc. All this information is contained in a command interpreter tied to an interrupt in control_isr() on the bottom left of Figure 2. But…

How do you add the host command interrupt ISR (that implements Play/Stop/etc.)?

How does it interact witht the XINT DMA interrupt (control flow vs. data flow)?

Who should have priority, controller subsystem commands or DAC “feed me” requests?

In other words if while running the decode() function thread, what happens if the DMA i2sisr interrupt comes in with the DAC asking for more “feed me” data?

Can the DSP miss a real-time deadline like that even though the system has plenty of MIPs?

The above questions are classically handled with custom preemptive kernals and interrupt handlers. Often the system is then laboriously hand tweaked for a given configuration. Then if any changes need to be made, the tweaking effort must be repeated.

Also if you want the DSP-only solution, adding the Controller subsystem side mentioned in Figure 1 gets very complicated. Now you have even more threads (storage media access, user if with keypad/LCD, DSP Subsystem data requests.

Finally, how do you debug effectively real time? These questions will be answered in the next section *Adding DSP/BIOS to the MP3 Player (DSP Subsystem)*.

4 Adding DSP/BIOS to the MP3 Player (DSP Subsystem)

Let us start to look at DSP/BIOS Objects, and how they were added to the MP3 software for the shipping systems TODAY. Figure 3 shows the existing DSP/BIOS objects, in alphabetical order:

	Module Description
6 – CLK	System clock manager
5 – GBL	Global setting manager
5 – HST	Host input/output manager
1 – HWI	Hardware interrupt manager
5 – IDL	Idle function and processing loop manager
5 – LOG	Message Log manager
1 – MEM	Memory manager
3 – PIP	Data pipe manager
4 – PRD	Periodic function manager
5 – RTDX	Real–Time Data Exchange manager
7 – STS	Statistics accumulator manager
2 – SWI	Software interrupt manager
5 – TRC	Trace manager

Figure 3. DSP/BIOS Objects Summary

We will use these DSP Bios objects to implement the control structure for the MP3 player. We will use the objects in the order of the number of the figures. Instead of boring you with a treatise and explaining each one, we will start adding them to the MP3 Player in Figure 2, and learn their functionality by example. For a more detailed explantation see C54x *DSP/BIOS User’s Guide* (literature number SPRU326).

The first step is pretty basic and involves getting the code into the Code Composer Studio environment. First, we create a Code Composer Studio project. (See the Code Composer User’s Guide.) Next, we add all the necessary source files into the project and let us call this step the zero degree. Now, we are ready to add the DSP objects.

4.1 HWI/(MEM) Objects (First degree)

The next step involves using the DSP/BIOS configuration tool to set up the hardware interrupt handlers. The configuration tool will generate a hardware interrupt vector table for you. If you wish, it will also add real-time analysis features to the hardware interrupts. Memory sections are thus available to the user “for free”. Using these objects does necessitate the use of a configuration file called the .cdb that is set using a graphical configuration tool . the .cdb creates assembly .s54 and .linker command .cmd files that are linked to make the .out executable).

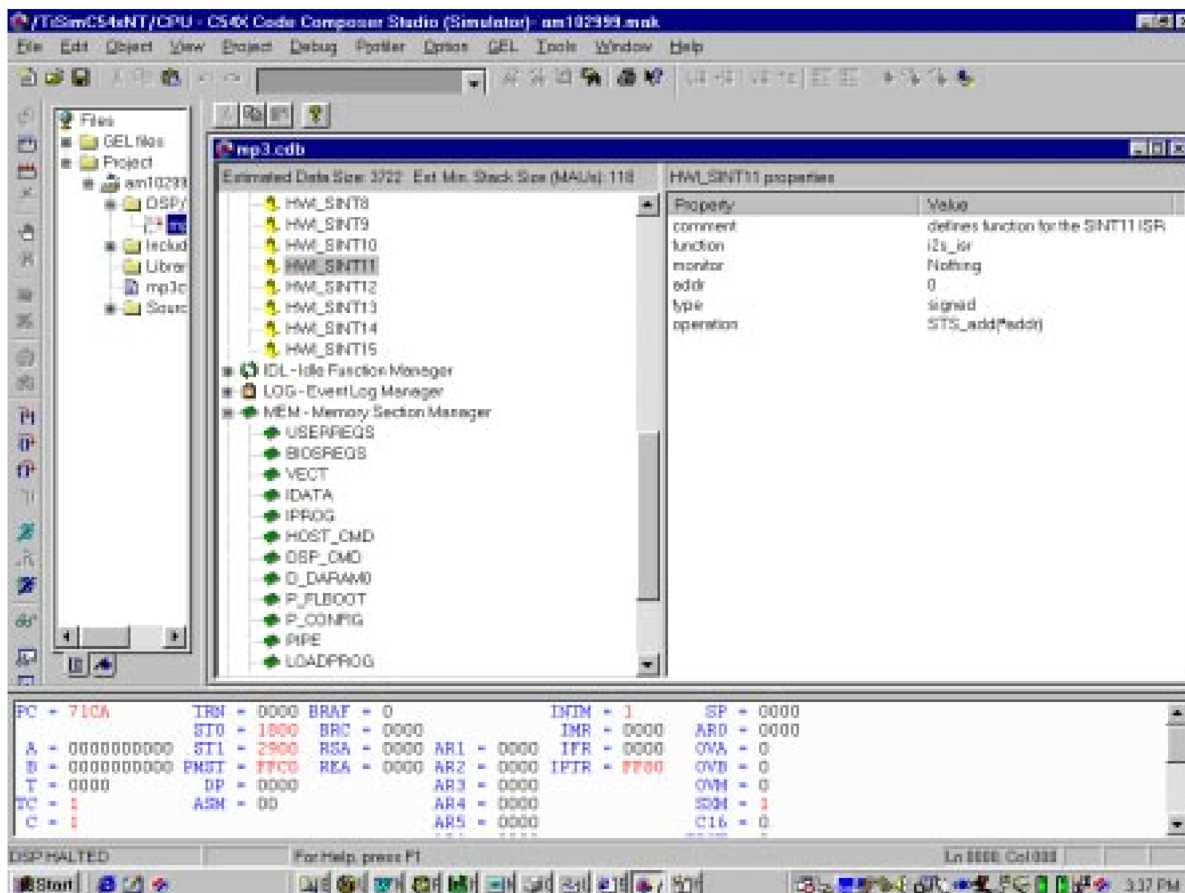


Figure 4. Configuration Tool Setup for HWI (/MEM)

Figure 4 shows a screen shot of the Code Composer Studio integrated development environment. The DSP/BIOS configuration tool window is open in the middle, with the window title “mp3.cdb”. The left hand side of the configuration windows shows the DSP/BIOS objects in your application. The right hand side shows the properties for a selected object. The DSP/BIOS objects are organized in a two-level hierarchy of *module managers* and *objects*. Managers contain settings that affect all objects they manage.

The DMA interrupt based on XINT, labeled HWI_SINT11 in the configuration tool, is shown selected in Figure 4. In the right hand side of the window we see that i2_isr is set as the function to run for this hardware interrupt. We must also configure the HPI DSP Interrupt, shown as HWI_SINT9 in the configuration tool.

Below the HWIs in the mp3.cdb window is the memory configuration objects and manager (that we get for “free” with the .cdb file). The MEM objects are replace the MEMORY command from a traditional linker command file. Then these MEMORY partitions can be used by other objects in the configuration tool and in the linker command file SECTIONS command.

Also, as promised in Section 3, the left side of Figure 4 shows the project window. This window contains associated files that source the MP3 Player, and are based on the mp3.mak project file.

With the added HWI objects, Figure 2 becomes Figure 5:

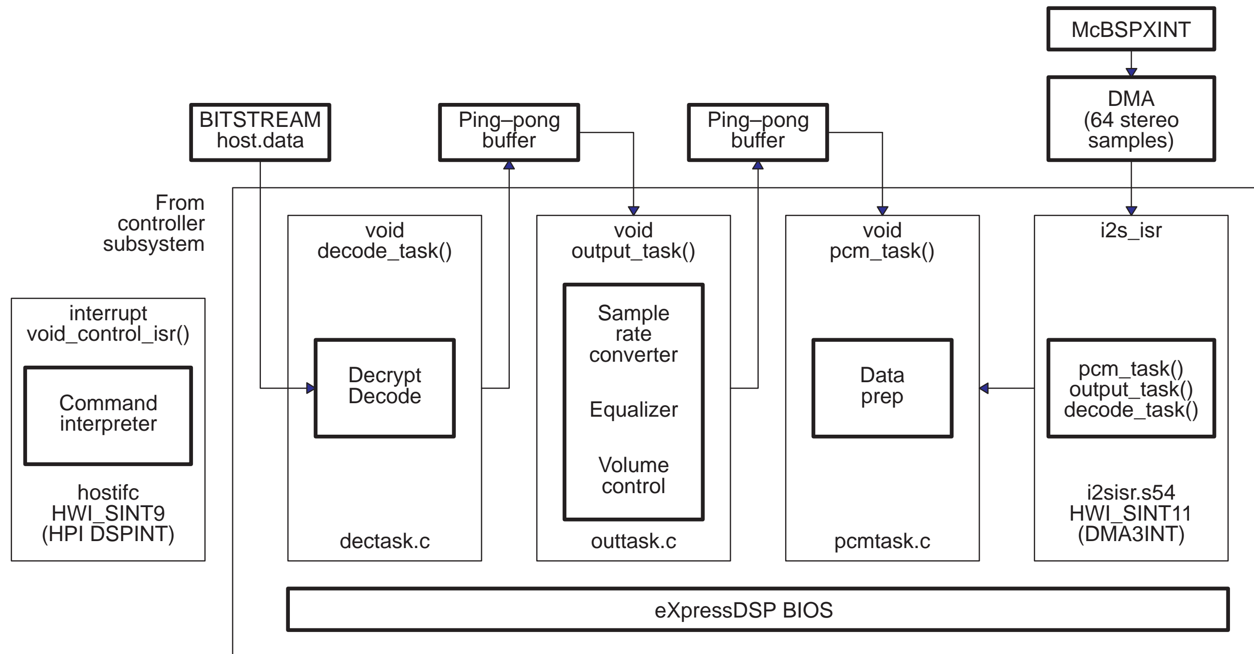


Figure 5. Adding HWI Objects to the DSP Subsystem

Note that the DSP Subsystem control/data flow looks prettier in Figure 5, but the system is identical to the system in Figure 2 with the same problems. Two HWIs are used, one for data to the codec/DAC and one for commands from the controller subsystem. Note that HWIs are the first level (and highest priority) of “threads” within DSP/BIOS. For more detail on threads, see C54x *DSP/BIOS User’s Guide* (literature number SPRU326).

The benefit of this First Degree is that your project is built within the Code Composer Studio project tool for convenience, and uyour hardware interrupt vector table is generated and placed by DSP/BIOS. By allowing DSP/BIOS to generate the hardware interrupt vector table, DSP/BIOS can transparently use the TI RTDX technology, which uses the JTAG emulator, to send real-time debug information to Code Composer Studio without stopping the target. The data sent in real time is called *Real-Time Analysis* data, and is displayed with a variety of Code Composer Studio plug-ins.

The DSP/BIOS LOG object and Message Log plug-in are an example of Real-Time Analysis. Instead of using an intrusive printf that can take 27K bytes of memory and 30K instructions per second, the LOG object (which will be discussed in more detail in Section 5.1) can be used to do a LOG_printf. This object takes only 32 bytes and only 36 cycles for debug (since all the work is done on the debugging PC). The LOG_printf’s can be (and were) used extensively at this stage for debug. They can be read in detail on pp. 6–16 of *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (literature number SPRA591).

Having said all this, you look at the two HWIs in the system and re-ask the question from Section 3.

Who should have priority, Controller subsystem commands, HWI, or DAC “feed me” requests HWI?

Should the user of the MP3 player see a large delay when they hit STOP or FFWD on their system, or is it acceptable for some of the music to drop out?

In other words the classic system tradeoff question, what has higher priority control or data?

So now let us get into the meat of the DSP/BIOS. We will go a little out of order compared to *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (literature number SPRA591), jumping from first to third degree.

4.2 SWI Objects (Third Degree)

In this section we do something new and use MULTIPLE threads, and hence the DSP/BIOS scheduler, instead of doing an entire MP3 player in one ISR. The HWI is split up into its logical threads providing more granularity for making decisions on system priority configuration. These threads are known as Software Interrupts or SWIs.

An SWI is a thread posted on a stack of SWI threads that need to be completed in priority to complete execution. The posting is done in code from a variety of ways including function calls, ISRs, HWIs, and some DSP/BIOS objects we will see in the rest of this document. Again for more detail on threads and the DSP/BIOS scheduler, see C54x *DSP/BIOS User's Guide* (literature number SPRU326). Also, as a minor point, note that the posting is only 1-deep like the IFR (Interrupt Flag) of the C5000/C6000™ DSPs. If an SWI is posted multiple times by different threads or objects, it will only be run through completion. Once the SWI is completed, it must be posted again to run. The acute reader may notice this effect as we go through the system.

SWI is set using the Configuration Tool. See Figure 6 for the SWI Manager and the ease of setting thread priorities, especially SWI.

C6000 is a trademark of Texas Instruments.

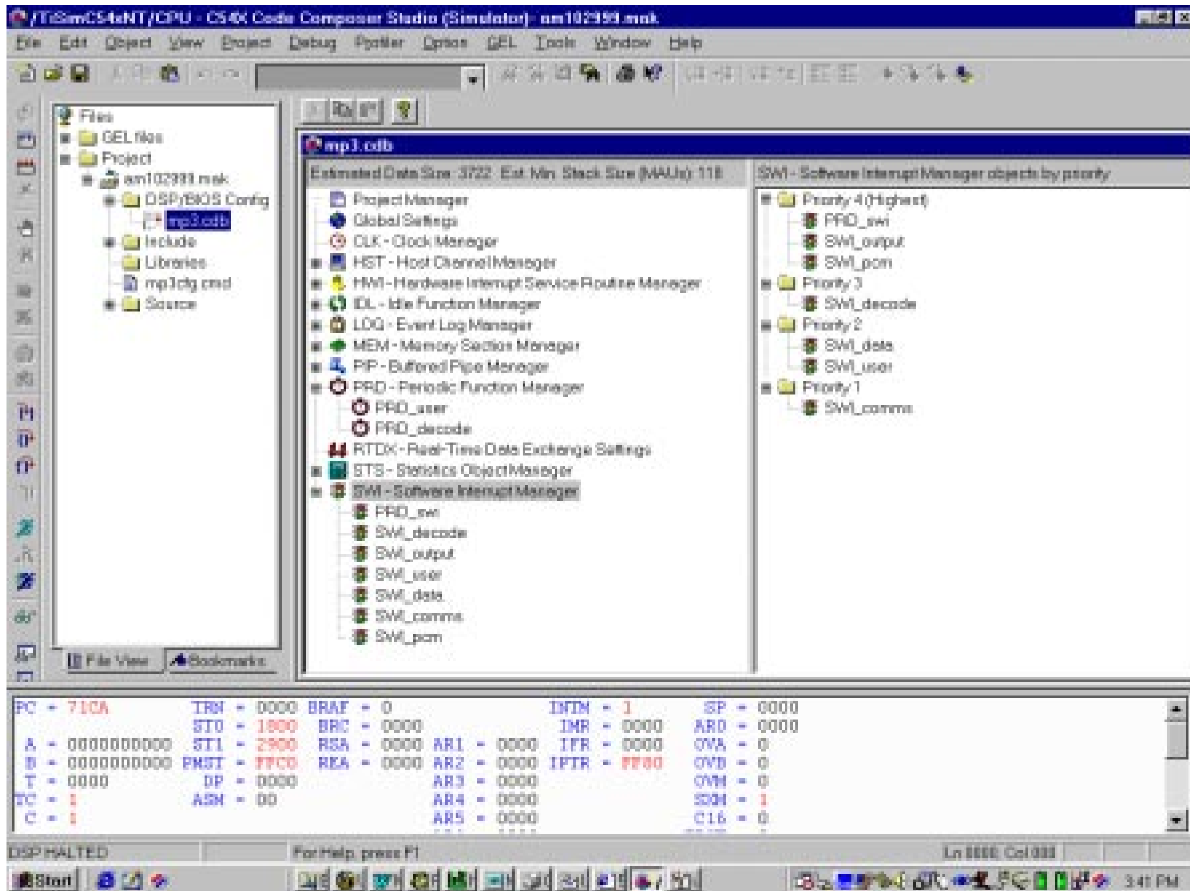


Figure 6. Configuration Tool Setup for SWI

On the left side of the mp3.cdb window, the SWI manager is selected. All the SWI objects are seen displayed (if you select an SWI object you can set the corresponding C or assembly function/ISR that is deemed similar to HWIs that we saw in Figure 4). We divided all the functions in the i2isr from Figure 2 and/or 5 into multiple SWIs. This gives more granularity in what priority each function/threads should be run.

On the right side of mp3.cdb the SWI priorities are easily set graphically by pointing, clicking, and moving with your mouse up to 16 different priority levels. SWIs at the same priority level will run until completion. SWIs with higher priorities will preempt those SWIs with lower priorities. The bottom line benefit is that no more complicated interrupt handlers are needed. DSP/BIOS easily allows the balancing of threads and their priorities. In Section 5, we will see how this balancing can be done real-time.

So now what does setting up such an SWI system, and hence using the DSP/BIOS scheduler do to our MP3 software from Figure 5? Let us look at Figure 7:

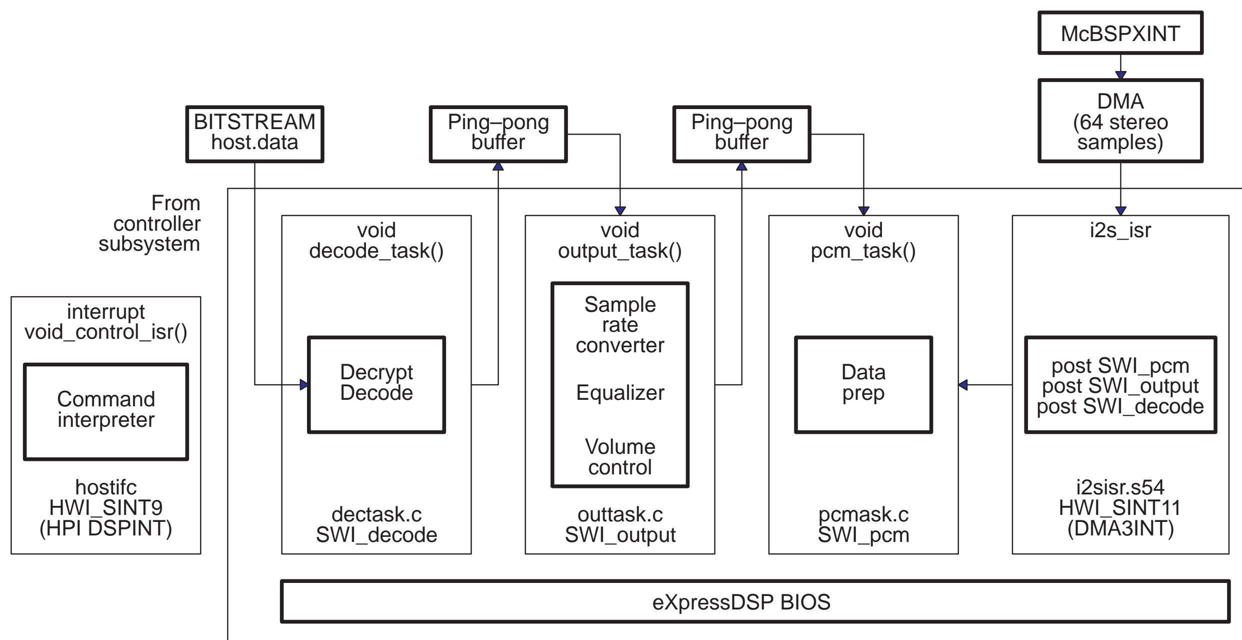


Figure 7. Adding SWIs to the DSP Subsystem

As mentioned earlier, the i2s_isr blocks have been split into 3 SWIs, SWI_decode, SWI_output, and SWI_pcm. The former two SWIs have identical function as the decode() and output() functions seen in Figure 2. The latter SWI was split out of the output() function to do some prepping of the data buffer and requesting more data (that will be covered in section 4.3). The DMA Interrupt based on 64 XINT interrupts/samples is still configured to call i2sisr. Instead of running all these functions in a huge ISR, each of the SWIs could be posted to run. Thus the SWIs can be run with more granularity with varying priorities. They may be debugged easily by changing in the SWI Manager and recompiling the code. The threads are run more efficiently in the “background” as appropriate. So HWI_SINT11 has highest priority to deal with the “feed me” nature of the codec/DAC, but the actual work is configurable real time.

The second HWI in the MP3 system is the command HWI. Note that the command interpreter in control_isr() is consciously not being posted, but still run as a classic ISR. In other words, HWI_SINT9 stays the same giving it highest priority. In this system, this configuration guarantees a maximum latency for an external controller to wait for a command (PLAY, STOP, FFWD, etc.) to the DSP system.

Could you partition the HWI with some SWIs to post at lower priorities for less important commands (like trace buffer info) and keep the more important commands in HWI?

Take a moment and reflect on the paradigm shift in using the DSP/BIOS scheduler. Each of the functional blocks or threads in the data path is now placed on the SWI priority stack. SWIs may be posted , even due to a HWI, and run in the background. The system is not following the typical mode (seen in Figure 2) of running off helter skelter to service an interrupt and preventing other ISRs from running and missing real-time operation. Rather the system is run in a balanced mode of threads, HWIs and SWIs, to give the most efficient, real-time operation.

Having said that, we still may use HWIs to give those threads the highest priority and minimize latency in the DSP/BIOS system where it needs to be balanced. Again i2s_isr for HWI_SINT11 (DMAINT3) minimizes locking out other HWIs (command interpreter) by merely posting SWI_pcm. Meanwhile the DSP Subsystem command interpreter HWI_SINT9 gets highest priority in the system, and it just happens to be a C interrupt ISR.

For your own system this balancing must be debugged and the DSP/BIOS Objects in section 5 will help you do that. Note that since control and data can be partitioned and mixed more efficiently now with DSP/BIOS, the possibility of doing more and more control code on the DSP becomes a reality.

We are not done building the MP3 system yet. We can make use of another DSP/BIOS service that will make our system easier to modify and observe. Right now our system uses ping-pong buffers to pass data between processing steps. There are two parts to a ping-pong buffer. There are the buffers themselves which contain the data being passed, and there are flags to tell the reader and writer which buffer to use (read from ping while writing to pong, read from pong while writing to ping). The DSP/BIOS PIP (data pipe) module combines these features into a single object.

4.3 PIP Objects

Let us put a thread data sharing structure into the DSP subsystem in the form of a PIP (pipe) object. A PIP is a statically defined buffer in memory (and you can define the memory by MEM objects) that allow threads to share data. The first thread is known as the writer of the PIP. It processes some data, writes to the PIP, and when it finishes writing the reader thread is posted to process the data. Again full detail can be read in C54x *DSP/BIOS User's Guide* (literature number SPRU326), but let us see how we configure PIP_decode as a data sharing structure between SWI_decode and SWI_output. See Figure 9 for the data flow model:

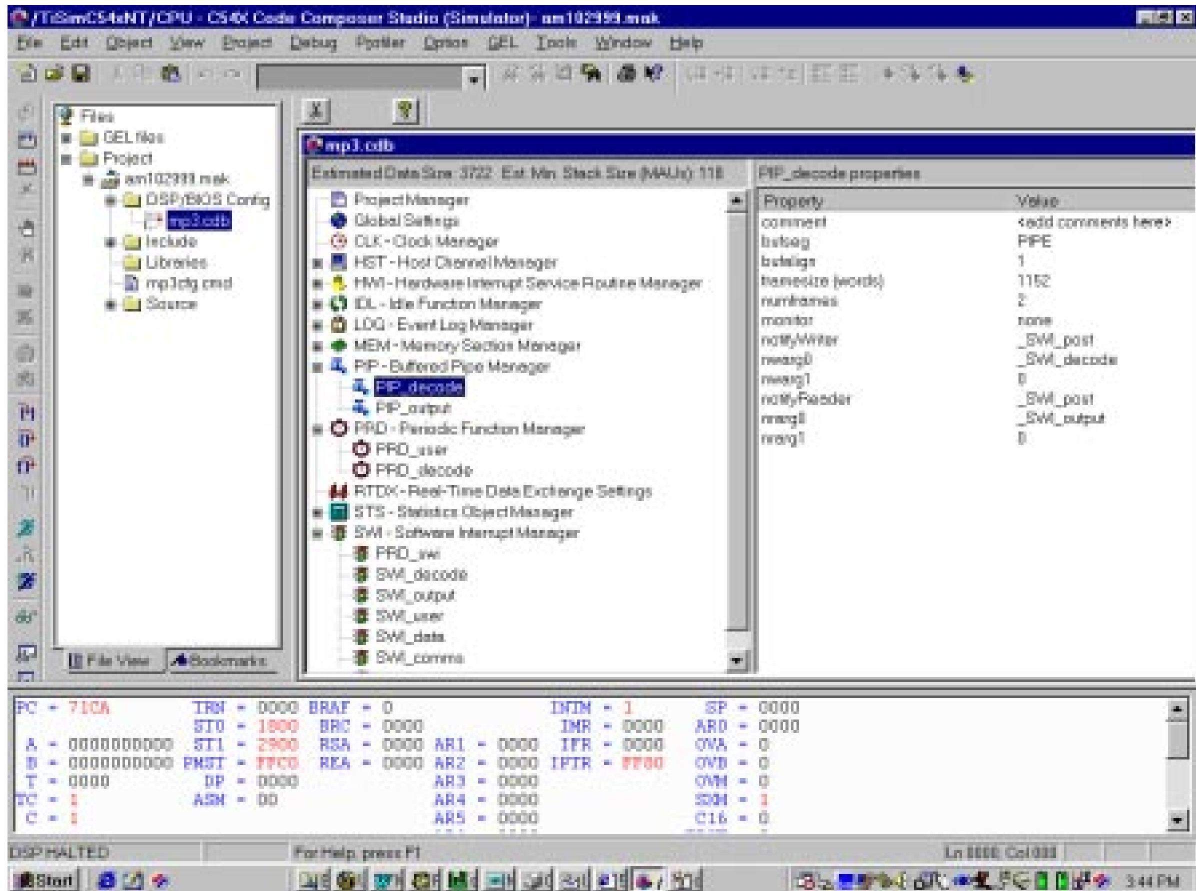


Figure 8. Configuration Setup Tool for PIP

Figure 8 again shows the DSP/BIOS configuration tool. We have created two DSP/BIOS PIP objects under the PIP manager, PIP_decode and PIP_output. PIP_decode is selected. Its properties are shown on the right side of the configuration tool. The notifyWriter and notifyReader properties have been configured to functions which post the reader and writer software interrupt threads when an empty buffer and full buffer are available.

Thus changing the ping-pong buffers in Figure 7 to PIP objects makes block diagram look like Figure 9.

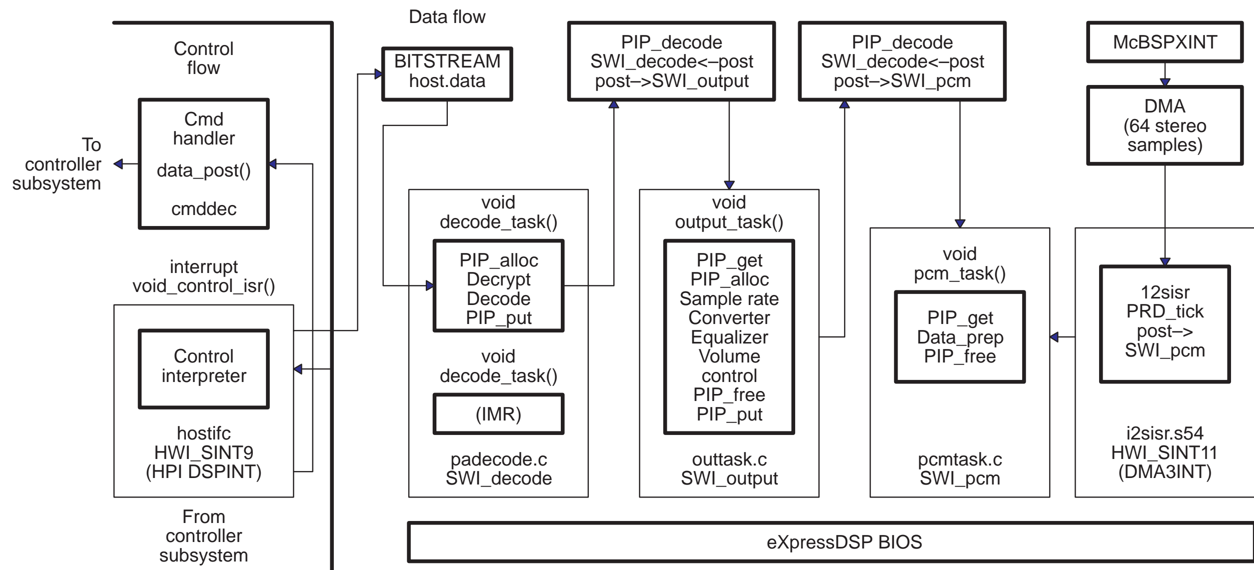


Figure 9. Adding PIPs to the DSP Subsystem

In Figure 9, PIP_decode and PIP_output have been substituted between SWI_decode/SWI_output and SWI_output/SWI_pcm, respectively. The names are arbitrary. In SWI_decode after a PIP_alloc function is used to bind PIP_decode and the decrypt and decode functions are run, PIP_put function indicates when PIP_decode is full. This function causes the appropriate notifyReader action of SWI_post of SWI_output to occur. After SWI_output empties and processes (SRC, EQ, VOL) the data (and happens to fill up another PIP called PIP_output), the PIP_free function executes the notifyWriter action of SWI_post of SWI_decode so that SWI_decode may start filling PIP_decode again. And as noted, a second PIP_output is used in a similar fashion.

In summary, the PIP object allows a more elegant way (i.e., API) of passing data between threads than a ping-pong buffer. To clarify a bit deeper though while a single-frame PIP is not a ping-pong buffer, a multi-frame PIP may be set up. A two-frame PIP by default is a ping-pong buffer, though of course it is more elegant a ping-pong buffer with a well defined API. That covers the data flow.

To complete the control flow, the DSP has a command handler that interrupts the Controller subsystem when it needs more data (coming from the SWI_pcm data_input_request() function). Meanwhile, the DSP Subsystem command interpreter is run based on an HPI DSPINT interrupt from the Controller subsystem and is folded into the HWI_SINT9

But we are not quite done for a full MP3 player. So far we have been adding DSP/BIOS objects to the DSP Subsystem only for academic and simplicity purposes. Implementation of PLAY/STOP/FFWD commands does not complete an MP3 player. User interface and storage media interface (i.e. Compact Flash, you have to store the music files somewhere) still need to be done. These functions are accomplished in the Controller subsystem.

Let us look at adding DSP/BIOS into the Controller subsystem, and also solve another real time problem with a new DSP/BIOS object called PRD. Note in Figure 9 under i2s_isr there is a PRD_tick function that will be used in the next subsection.

4.4 Adding the Controller Subsystem (and the PRD object) to the MP3 Player

As mentioned in the last section, we need to add user interface and storage media capability to the DSP Subsystem to use this system as a stand-alone MP3 Player. DSP/BIOS helps allow control code to be efficiently ported to the DSP, allowing for a single processor system. Thus we will add the Controller subsystem functions that were mentioned in section 2.2. This code is simpler than the DSP subsystem. We have already added the HWI and SWI objects mentioned previously in this section, and will cover them first. Let us look at Figure 10 from left to right, where as opposed to the DSP subsystem we will look at control flow first and then data flow.

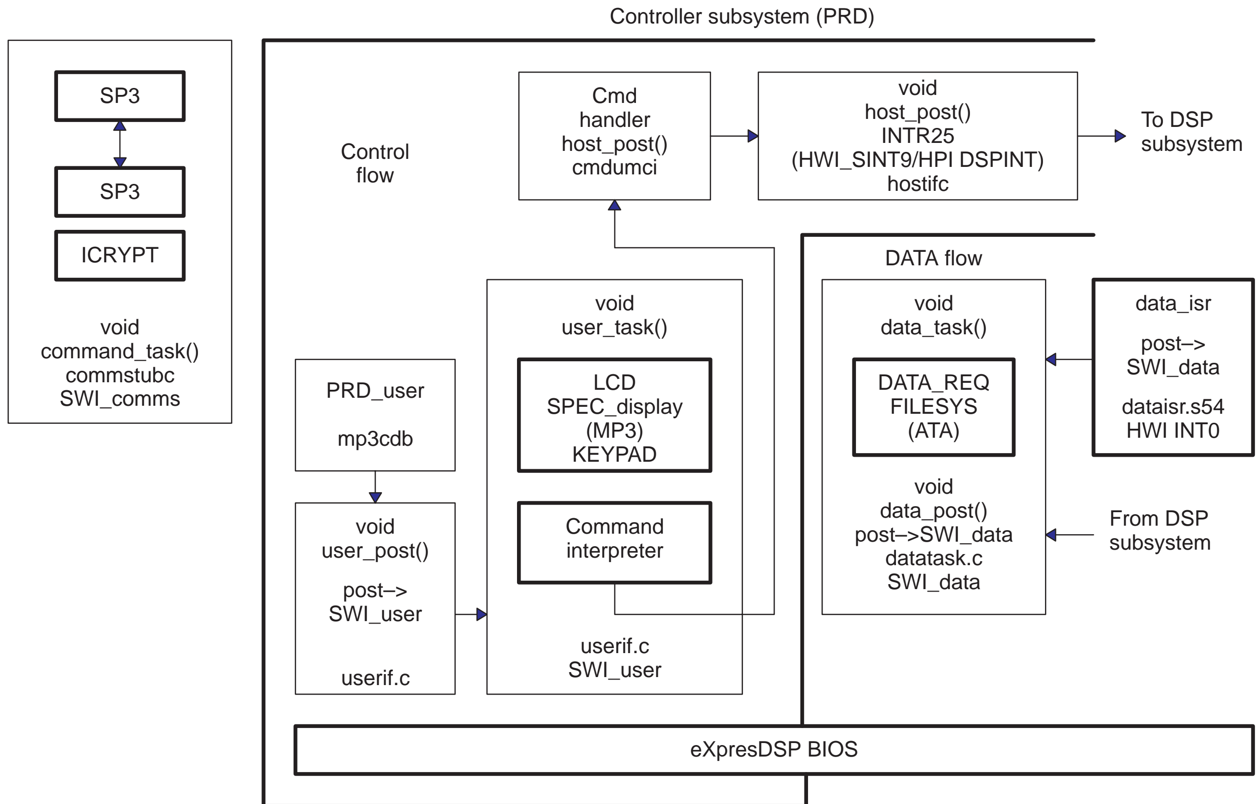


Figure 10. Controller Subsystem

The first block you see on the left side is SWI_comm which is used for secure downloads from the host whether PC or PDA. At this juncture this SWI runs separately and will not be discussed in detail. Basically it is a stub that will download music when the rest of the player is not running (though concurrent operation may be a future feature) that uses the PC serial or parallel ports or even USB.

In the middle of Figure 10 is SWI_user. This SWI is the user interface thread that indicates the beginning of the Controller subsystem part of the code. This thread runs the user_task() function that updates the LCD which might include spectrum analyzer info for MP3 files. It then checks the keypad for any commands from the user, and then a Controller subsystem command interpreter (Stop, Play, FFWD, etc.) executes to interpret commands from the user and sends a command to the DSP Subsystem. The command is sent using the host_post() function in the Controller subsystem command Handler that sends a HPI DSPINT interrupt to the DSP (as seen on the upper right part of Figure 10).

So the question arises, how often should the Controller subsystem user interface be invoked and how should it be invoked? Would it be nice to have a built in system “alarm clock” to schedule the user interface thread?

This functionality is done by using the DSP/BIOS PRD object. Back in Figure 8, the PRD Manager was opened under the PIP Manager. We defined in that manager PRD_user that is basically a periodic function set to an arbitrary number of ticks for about a 15 ms periodicity. Thus in Figure 10, the PRD_user thread goes off like an alarm about every 15 ms. The user_post() thread is called that does an SWI_post (at the appropriate priority in the SWI Manager) of SWI_user. So DSP/BIOS PRD objects can be used throughout the system as “alarm clocks” to run appropriate threads. In summary, PRD_user is just a 15 ms timer that invokes the function user_post() that posts SWI_user.

But this brings up a subtle question. Where is the system clock tick that allows the PRD to count?

There is a CLK object that we will discuss in section 5.2, but this was not chosen for reasons that will be explained there. Rather referring back to Figure 9 in the i2s_isr under the HWI_SINT11, a PRD_tick DSP/BIOS function was set up. This configuration allows the PRD system “tick” to be synchronized to the “feed me” codec/DAC interrupt. Technically one PRD_tick comes out (1/44.1 kHz) * 64 samples = 1.45 ms. Since PRD_user is set to 10 ticks, the “alarm clock” goes off every 14.5 ms or about 15 ms. Thus, in systems where external periodic signals already exists, the on-board DSP timer does not need to be used as a clock for the system.

That covers the control flow. Now let us discuss the data flow:

Whenever the DSP Subsystem needs more data (based on the codec/DAC “feed me” architecture that flowed back from HWI_SINT11 to the DSP Subsystem command handler), the DSP Subsystem interrupts the Controller Subsystem using the HWI_INT0 external interrupt that invoked the data_isr ISR. As can be seen on the bottom right side of Figure 10, data_isr would post the SWI_data software interrupt and let it run in the background based on SWI Manager priorities. The SWI_data thread needs to be invoked to access music data from whatever storage media (Compact Flash in the case of the IA EVM). This thread handles the data section of the Controller subsystem. As mentioned above it is invoked by an interrupt to the controller from the DSP. The HWI_INT0 is set up in the .cdb (similar to what was covered in subsection 4.1 for the DSP Subsystem) which calls the data_isr assembly ISR. This ISR posts SWI_data. The nice aspect of SWI_data is that this thread can independently access the storage media using ATA function calls while blindly following the data and sizes that the DSP subsystem requested.

We will summarize this entire MP3 player system, Controller and DSP subsystem, in section 5.4. But before we do, let us expand on some of the debug features that DSP/BIOS gives us. This involves the Second Degree of *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (literature number SPRA591).

So that is the entire code build. Now, how about debugging in real time?

5 DSP/BIOS RTA (Second Degree – with Implicit Instrumentation Objects)

DSP/BIOS Real-time Analysis (RTA) is a very powerful set of instrumentation tools that allow you to see and debug your system as it runs in real time. In section 4.1 we showed the LOG object and the LOG_printf function. However, DSP/BIOS includes many more features which require no code modifications whatsoever.

DSP/BIOS provides many forms of IMPLICIT instrumentation, that is you get the instrumentation just by using the kernel. No explicit calls like the LOG_printf shown in section 4 are required for these features. In other words, you get it for free without ANY code modifications past what we have done in Section 4. For much more detail on many other instrumentation permutations beyond what is shown in this section, refer to the C54x *DSP/BIOS User's Guide* (literature number SPRU326). There is a lot more you can do by just adding a few lines of DSP/BIOS functions in code.

5.1 Execution Graph and TRC/LOG/HST/RTDX/IDL Objects

There are a variety of DSP/BIOS Objects that enable RTA. The C54x *DSP/BIOS User's Guide* (SPRU326) will explain them in detail. But for a brief, layman's overview:

GBL – Sets global parameters for the system in which "enabling RTA" automatically adds the objects below.

TRC – Interesting events (usually DSP/BIOS objects like SWI, PRD, etc.) worth looking at.

LOG – Collects events in real time (like LOG_printf's and TRC) for sending target to debugger host (PC).

HST – Manages streaming data between data and debugger host (PC) over a generic channel.

RTDX – Manages a specific JTAG scan-based emulation channel for doing an HST.

IDL – Manages the time when the DSP/BIOS scheduler is doing no work and lets TRC/LOG/HST/RTDX to slip in.

So what does all this explanation look like graphically (a picture is definitely worth more than all the above words)? Figure 11 is an example of "RTA for free" (i.e. implied objects) while the MP3 player is running.

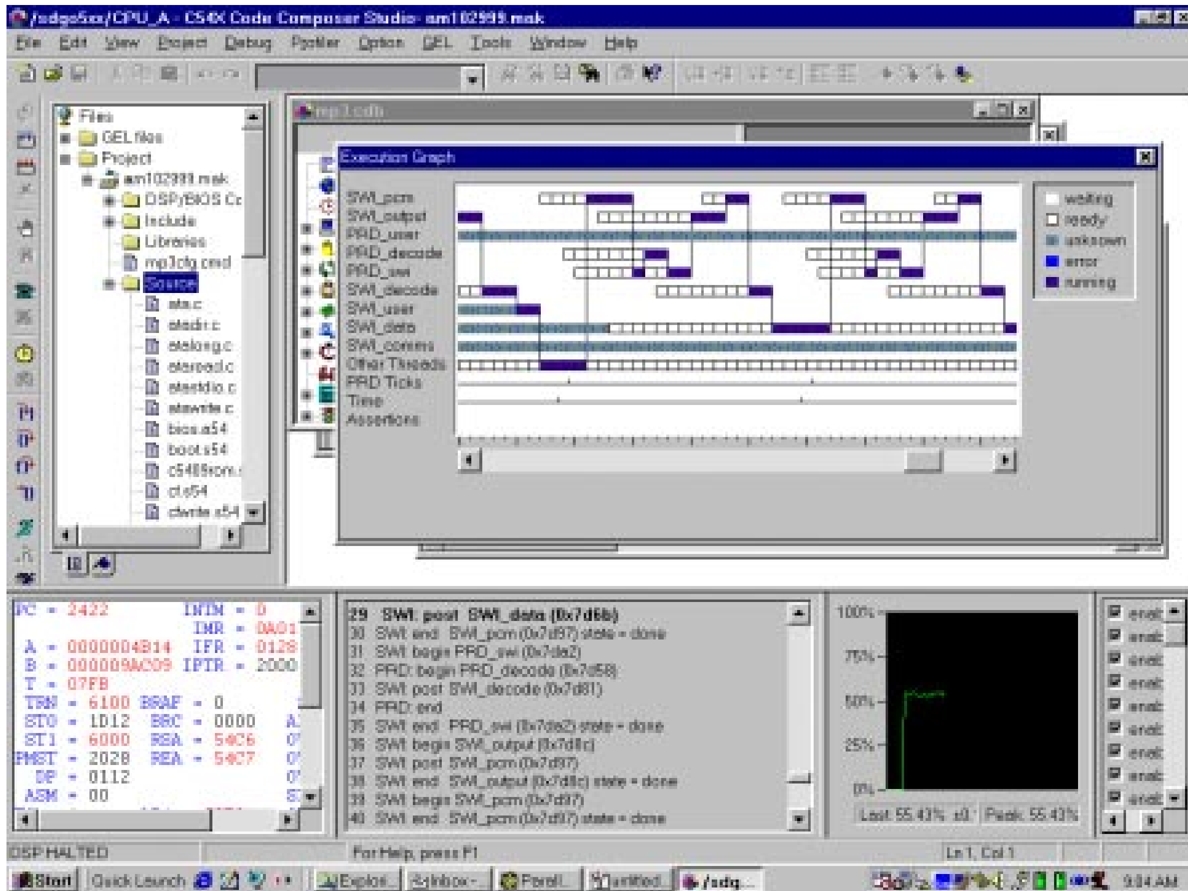


Figure 11. Brief DSP/BIOS RTA (Execution Graph/Log and Load Graph)

In the center of the CCS1.x screen with DSP/BIOS RTA is a typical Execution Graph with accompanying Message Log in the center below it. The Execution Graph graphically displays the information/events that is contained in the Message Log. DSP/BIOS implicitly supports various DSP/BIOS objects such as SWIs, PRDs , etc. all the SWIs discussed can be seen looking at the y-axis of the Execution Graph. The threads are colored based on when they are posted, run, and completed.

In this example we can concentrate on the data flow/music playing. The SWI_user seen at the left was processing when the user hit "play" on the keypad. That is why the SWI_data became known, and was posted to begin accessing the Compact Flash (the other DSP Subsystem threads are running, but do no real work before you hit "play"). The user clicked on the first white block of SWI_data, and that corresponds to the bolded "#29 SWI:post SWI_data" in the Message Log below. You may then follow some PRD activity until the beginning of SWI_output which was posted before #29. Since it has higher priority than the other SWIs (except SWI_pcm, which is the same priority. See Figure 6) it runs to completion. Though in the process, SWI_pcm was posted and it also runs to completion since it has the same priority as SWI_output and thus higher priority than the other SWIs (At this point we are at #40 in the Message Log, and getting off-screen on it). But on the Execution Graph you can see SWI_decode, which has higher priority than SWI_data, run to completion. And then finally SWI_data starts to run, but is pre-empted by SWI_pcm and the cycle continues.

Note the PRD_ticks at the bottom of the Execution Graph. Remember from section 4.4 that these ticks occur about every 15 us. It is important to note that the PRD_ticks show time, but not the blocks in the Execution Graph. They are showing DSP/BIOS events occurring, and are thus event based. We will look at real-time deadlines and benchmarking in subsection 5.3.

Before we move on though, we should look at the UNIX looking box in the bottom-right hand corner of Figure 11.

5.2 Load Graph and the CLK Object

In the bottom-right hand corner of Figure 11 is a classic CPU load graph. Again if you refer to C54x *DSP/BIOS User's Guide* (literature number SPRU326), there is much detail in how this number is calculated. But basically DSP/BIOS "keeps track of when" it is in the IDL object (idling and doing no real work, which includes RTA work) and subtracts that MIP number from the total MIP's. Note that the load right spikes from near 0 to 55%. This indicates the user pressing "play".

For DSP/BIOS to "keep track of when" as mentioned in the previous paragraph, it needs a clock. There is a CLK object that runs off a C54x timer. The CLK object can be used to clock source the PRD objects, even though we chose to not to in this system and used the i2s_isr PRD_tick instead.

Another benefit of the CLK Object is that it time stamps the Execution Graph/Message Log. If you look ahead to Figure 12, the Execution Graph Message Log shows a CLK time stamp #23 right after the posting of SWI_data (#19).

Moving on to the next section, how about automatically seeing the real-time deadlines and benchmarking the threads that are being run in DSP/BIOS?

5.3 Real-time Deadlines, Benchmarking, and the STS Object

Using the profiling functions of Code Composer Studio (which used JTAG scan-based emulation hardware timers) is still viable for benchmarking functions/threads, but it cannot be done real-time.

For real-time system analysis, we can use the STS objects within DSP/BIOS. Figure 12 shows another MP3 player running screen shot. But this figure has more RTA detail, including the STS real-time deadlines and benchmarking statistics view:

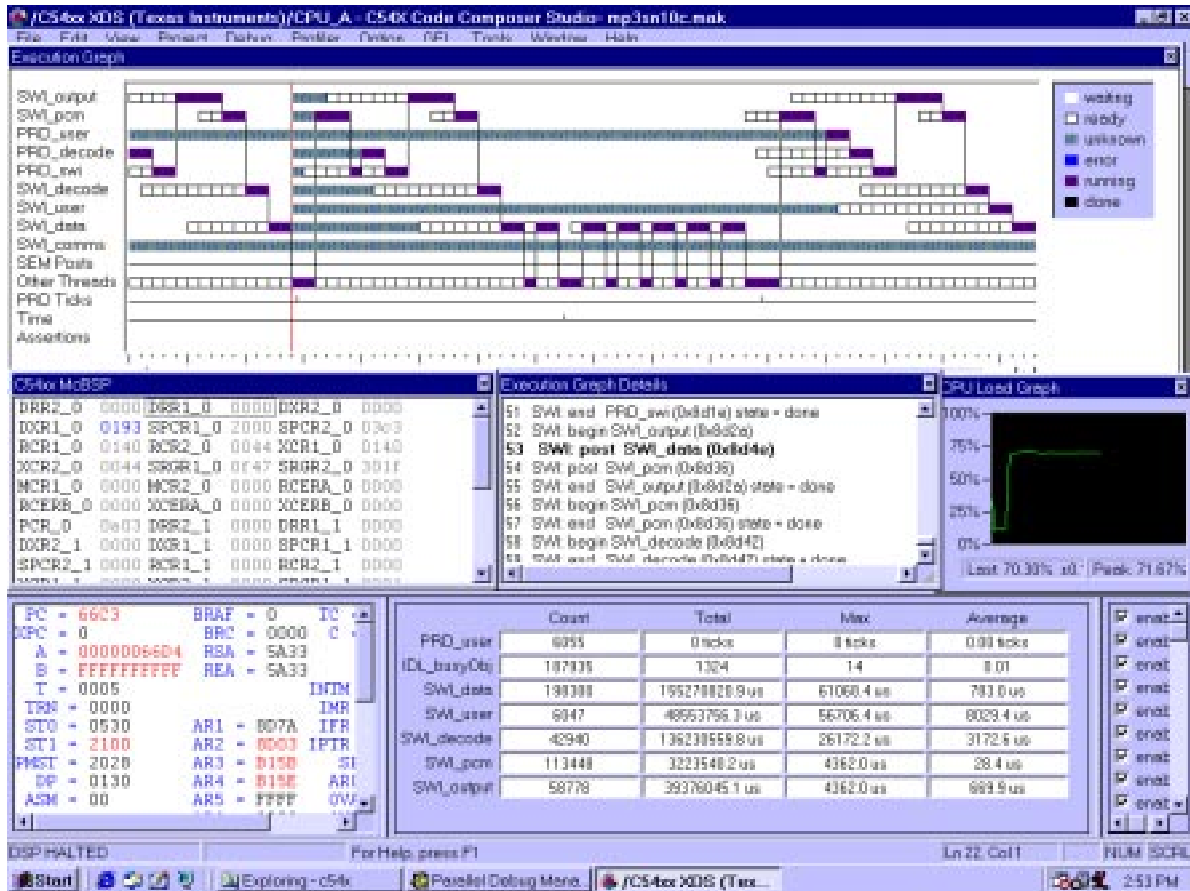


Figure 12. Detailed DSP/BIOS RTA with STS Accumulator

Again if you refer to C54x *DSP/BIOS User's Guide* (literature number SPRU326), at lot more detail will be provided regarding the STS Objects and Statistics view. For Implicit statistics (the one you get automatically), HWI, PIP, PRD, and SWI can all be enabled (see bottom right corner of Figure 12 where they are all checkmarked). But for simplicity in this example, we are basically only displaying SWIs.

Thus looking at the Statistics View window the various SWIs are seen displayed along with a four different columns. The last two columns are the most important. The Average column displays the Average (Total/Count) time spent for that SWI thread to go from **post to completion.** Thus it tells you on average if your SWI thread is at high enough priority to make real time, on average. The max column tells you what the maximum time was spent for that SWI thread to go from **post to completion.** Thus, it tells you on if your that thread EVER missed real-time (which in the case of the MP3 Player, the codec is muted to prevent any noise being heard by the user.

As you can see in Figure 12, each SWI thread has a max and average listed in microseconds. Let us now move to the next section 5.4 to summarize the entire system. The Max and Average numbers will be addressed towards the end of the section.

5.4 MP3 Player System Real-time Thread Summary

Figure 13 is a simple summary diagram of all the different threads in the system.

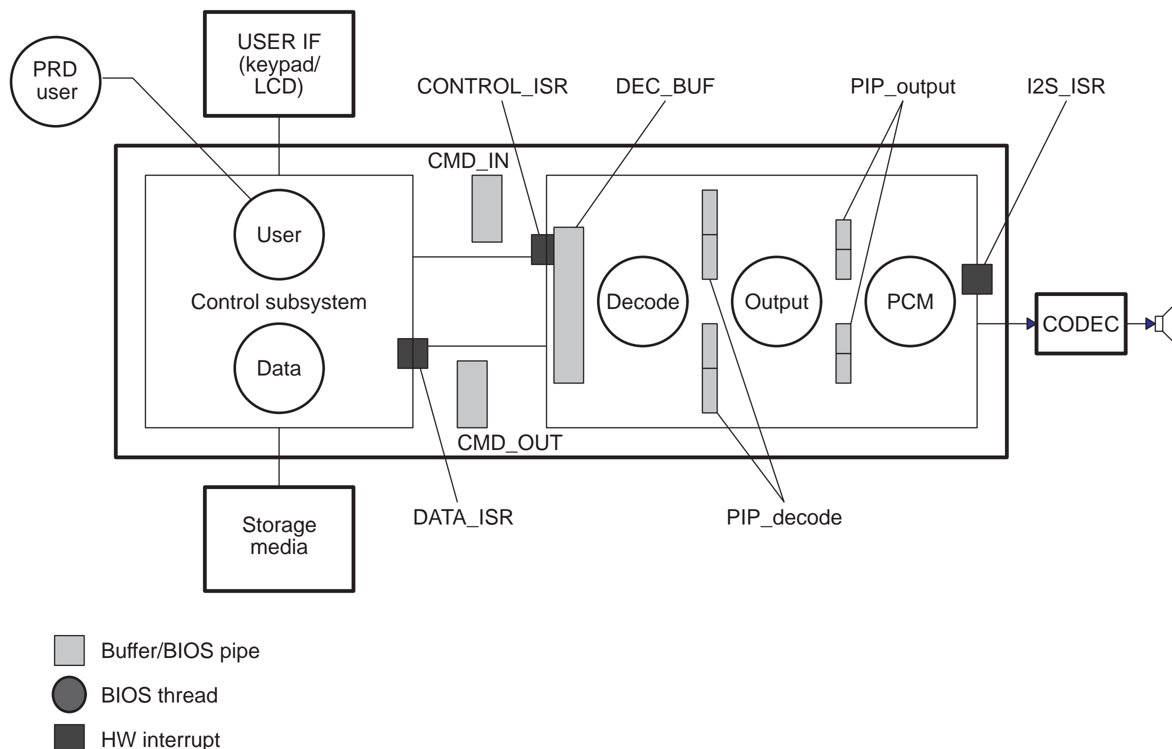


Figure 13. Simple Real-Block Diagram Summary of MP3 Player Threads

The “feed me” demand from the DAC ripples back to through the threads. SWI_output and SWI_pcm are given the highest priorities in the DSP/BIOS kernel to handle the data feed management. Since the scheduler is non-preemptive for threads of the same priority, one will have to complete before the other is begun. SWI_decode is done at the next priority in the DSP/BIOS scheduler and can vary in how much of the frame it decodes at a time based on something I still need to figure out. Also if it is running and SWI_output or SWI_pcm are posted, SWI_decode will be pre-empted according to the function of DSP/BIOS. SWI_decode could be viewed as running in the background of the above mentioned threads.

SWI_user was arbitrarily set using a PRD_user object in the .cdb file at 15 ms to update keypad and LCD. SWI_data will depend on the storage media used, but on Amidala with Compact Flash and 128 kps MP3 is running about 10% of the entire 44.1 khz duty cycle (i.e., 2 us or 4 kHz). This will greatly vary with your storage media and some with your decoder. Note that SWI_user and SWI_data are both placed at the same priority, but at a lesser level than even SWI_decode. Thus those threads are running even more in the background and will be the first ones to fail if you are violating real time in your system (i.e., do not have enough MIPs for what you are trying to do).

Again one interesting point to note is how SWIs may be posted, even due to a HWI, to give HWIs the highest priority and minimizes latency by increasing thread granularity in the DSP/BIOS system. Note how in the i2s_isr for HWI_SINT11 (DMAINT3) minimizes locking out other HWIs by merely posting SWI_pcm. Thus the DSP Subsystem command interpreter HWI_SINT9 gets highest priority in the system since it is a C interrupt ISR.

Table 1 is a simple summary chart of all the different threads in the system, their priorities, and when they are run. It nicely coincides to the block diagram in Figure 13.

Table 1. Real-time Summary of MP3 Player Threads

Name	Purpose	Runs When?	Priority	Frequency	Real-Time
Controller Subsystem					
SWI_user	Monitor keypad commands and update LCD	PRD_user object or Host timer	2 low	15 ms	15 ms
SWI_data	Access requested data from storage media	data_post() or host into from DSP	2 low	Approx every 64 samples	1.4 ms
DSP Subsystem Control					
HWI_SINT9 (HPI DSPINT)	Service commands and input data deliveries from host.	INTR 25 or int. from host.	Highest	Not more than once every 10 ms	1.4 ms
DSP Subsystem Data					
SWI_decode	Decrypt and decode audio bitstream.	Post form PIP_decode.	3	Every 64 samples. Decides evert 64 to 1024 samples.	13 ms (MP3)
SWI_output	Run SRC, EQ, VOL.	Post from output and decode	4 high	Every 64 samples	1.5 ms
SWI_pcm	Setup DMA. Request input data (Run_data_input_req()).	Post from i2s_isr and PIP_output	4 high	Every 64 samples	1.5 ms
HWI_SINT11 (DMA3INT)	Start SWI_pcm	DMA interrupt	Highest	Every 64 samples	1.5 ms

Note from Table 1 that the bottom four threads refer to the data path in the DSP subsystem and can be followed bottom to top as Figure 13 can be followed right to left. The middle thread is the control thread for the DSP subsystem (and seen in the middle of Figure 13). Its period is based upon commands from the controller subsystem, though a response latency of 100 us is guaranteed by it being a HWI. In other words, the DSP/BIOS scheduler allows hardware interrupts to preempt all SWIs. Thus, the only other HWI with higher priority than this HWI_SINT9 (HPI DSPINT) are those with higher priority in the C54x CPU such as reset. The top two threads coincide with the controller subsystem, and they can be seen in Figure 13 on the left side.

Finally regarding the real-time nature of the system and whether the priorities are set correctly so that each thread makes their real time deadline, each deadline needs to be approximated. The right-most column on Table 1 displays these calculations to be compared with the real numbers shown in Figure 12.

Let us derive the easier ones first. The bottom three threads on the chart are all need to run around the DMA3INT time coming from the DAC. In section 4.4 we derived this as 1 454 us, so let us say 1.5 ms. If you look at the bottom two SWIs, SWI_pcm and SWI_output in Figure 12, the Average times are 28.4 us and 669 us. These values are well within the 1.5 ms in Table 1. Now jumping to the control side in the same Figure, SWI_user is shown as 15 ms. This value is also clearly derived in Section 4.4 as 10 PRD_ticks. Looking at Figure 12, SWI_user Average shows 8 029 us which is within 15 ms.

The Average numbers in Figure 12 come from a steady-state operation of the system (in other words playing a song on the MP3 player). The operation can be easily observed as the step in the load graph

Anyone notice that the Max's in all the SWI do violate real-time?

This state occurs at initialization. All you need to do is clear the STS window after playing the song to get a real max. The previously mentioned SWI threads all had an obvious periodicity.

When a song is playing, SWI_decode and SWI_data are running based on the difficulty and characteristics of the particular song. MP3 makes use of "bit reservoirs" since it is a time-variant compression, unlike the vocoders more typical in DSP systems. This feature means that extra bits that need to be compressed for an MP3 frame may be borrowed from a frame before or after the one playing for a difficult part of the song. Thus the frame being decoded might not necessarily be the one playing for that slice of time. Since this "borrowing" occurs over only a few frames, SWI_decode can be approximated directly to the same rate based on the input buffer size. (The DSP actually handles reordering of the data in DEC_BUF on Figure 13 based on the amount "consumed" by SWI_decode.) Thus:

(1/44.1 kHz) * (DEC_BUF size) = 22 us * 576 stereo sample = ~13 ms

From Figure 12 the number is well within the 3 172 us number that DSP/BIOS STS reported.

Finally, SWI_data runs to completion based on the data requests that come from the DSP subsystem based on the song decoded by SWI_decode. Since SWI_decode requests are mainly based on the bit rate of the song (with some inaccuracy due to the bit reservoir), the real-time requirement can be approximated as the data rate over the HPI converted from bytes to stereo sampled words multiplied by the buffer size. Thus doing the conversions from bandwidth to time make the real-time deadline:

= 1 / [(128 kbit/sec) * (1byte/8bits) *(16-bit word/2 bytes) * 2 stereo channels/16-bit) * (DEC_BUF size)]

= 1 / [(128k/32) * (DEC_BUF size) = 1 / [4 kwords/s) * (576 stereo sample)]
= 1440 us = 1.44 ms

Again, as seen in Figure 12, this number is within the 783 us number that DSP/BIOS STS reported for SWI_data.

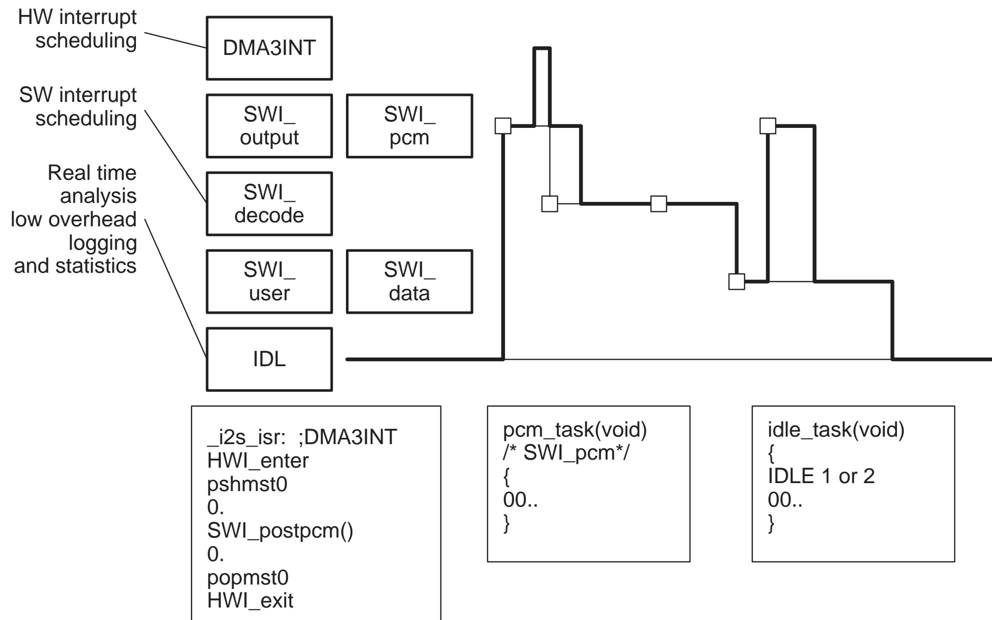
Thus, you see how the STS can be used to make sure a function runs within its real time. The methodology here is rate monotonic scheduling where you place the threads with the highest rates at the highest priorities (with SWI_decode being the exception). Of course, it is boring to see this working system. It is more interesting to change SWI priorities and see the numbers on a broken system.

So let us now look at some other benefits of the DSP/BIOS system.

6 Other Benefits

DSP/BIOS in ROM – C5000 devices could be manufactured with the DSP/BIOS macros burned into the ROM so that the user does not need to use SRAM for putting the libraries.

Low Power in IDL – While in the DSP/BIOS IDL loop, a C5000 IDLE instruction may be executed to save power (power management). This concept is illustrated in Figure 14:



Do with C54x IDLE instruction in DSP/BIOS IDL object.
– IDLE 1 with DSP only (peripheral i.e. timer intrrupt will wake BIOS).
– IDLE 2 with external controller/logic (INTx will wake BIOS).
– IDLE 3 requires external clock control along with IDLE 2 conditions.
RTA numbers will not be accurate due to IDLE.

Figure 14. Power Management in DSP/BIOS

This figure shows interaction of the entire system from a timing/execution graph view of threads on the top. The HWI for DMAINT3 is the highest priority. Below that are the SWIs at different priority levels. Finally we have the IDL. The corresponding code segments are for each thread and are shown at the bottom of the figure.

But let us concentrate on the IDL. Thus an IDLE 1 instruction is placed in the IDL_idle_task Object. If you refer to SPRU326, the idle_task() is run when no other threads (including RTA) are run. Since the IA code is based on the DMA3INT (from McBSP0 XINTs), this interrupt will wake the system out of IDLE 1, do any IA and RTA work, and settle back to IDLE 1.

If you change this to IDLE 2 on a stand-alone IA EVM system, the system will fail (get stuck at the IDLE 2, hear no music) since the peripherals (McBSP and DMA are shut down). Thus, we could get no farther with IA EVM. If you had a split micro+DSP system then you might be able to get to IDLE 2 while playing music if an external micro wakes up the DSP. But it would have to be around every 22 us. Obviously IDLE 2 or IDLE 3 could be used with a micro if you wanted a ”sleep mode” where music was not being played.

Another way to save power is to slow down the clock to the minimum MIPs needed by the application (say 55 MIPs in the MP3 example). Integrating under the current curve will show that power may often be saved in this way.

eXpressDSP™ – All the software features brought by this product allow fast-time to market on known systems. This features include not only code generation and project management capabilities, but also DSP/BIOS with RTA and a DSP Algorithm Standard that allow easily mixing and matching available algorithms.

7 Conclusion

Hopefully this discussion of an MP3 player system that is based on DSP/BIOS and actually being used in shipping production systems makes the DSP/BIOS and eXpressDSP™ concepts more clear. This software system is in production today. The combination of data and control on a single DSP should be made easier with the possibility of adding more granularity to threads and their priorities. RTA makes debug a breeze.

Thus, the advantage of DSP/BIOS in your system and ease of adapting is high.

eXpressDSP is a trademark of Texas Instruments.

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.

Copyright © 2000, Texas Instruments Incorporated