TEXAS INSTRUMENTS

Application Report
SPRA782 - August 2001

How to Get Started With the DSP/BIOS Kernel

Andy Thé — *Software Field Sales*
David W. Dart (updated by Shawn Dirksen) — *Software Development Systems*

ABSTRACT

DSP/BIOS™ is TI's real-time embedded kernel for the TMS320C5000™ and TMS320C6000™ digital signal processors (DSPs). Understanding how to build applications using the DSP/BIOS kernel requires adopting a multi-threaded design paradigm that at first seems foreign for new users of DSPs and veteran DSP developers alike. However, once understood, designing DSP/BIOS applications becomes straightforward, and you will find developing applications easier to design, debug, maintain, and extend. A real-time kernel such as DSP/BIOS helps to factor the problem and facilitate a robust design that is easily maintainable.

Traditionally, DSP applications were very simple, typically using a single program loop to manage the required processing. Over time, DSP applications began to require concurrent processing, as applications demanded the DSP to perform more functions. Applications now typically require the DSP to do several things at once and at different rates. In addition, DSP applications typically change over time, requiring support for additional or modified features. Building modern DSP applications using traditional program loop paradigms is very challenging, difficult to maintain, and even more difficult to extend. Using the DSP/BIOS real-time kernel offers developers the foundation to build modern applications from simple to complex multi-threaded and multi-rate applications.

Building modern DSP applications requires adopting a design paradigm, which includes multi-threaded, preemptive, event-driven threads. This requires understanding of how to architect applications using one or more execution threads rather than processing loops. The DSP/BIOS kernel provides three distinct classes of execution threads with different execution, preemption, and suspension characteristics to build applications on. In support of these threads are several additional kernel objects that provide device-independent I/O, inter-thread communication and synchronization, and other real-time services.

This paper addresses how to get started building DSP applications using the DSP/BIOS kernel. The primary focus is to understand the multi-threaded design approach using DSP/BIOS components. This includes organizing and structuring an application around DSP/BIOS execution threads.

Contents

List of Figures

1 Introduction

By now, you have reviewed the *DSP/BIOS Technical Overview* (SPRA646) and other DSP/BIOS technical documentation, and you may be thinking about how you can use the DSP/BIOS kernel in your application.

For developers unfamiliar with real-time multi-threaded kernels, architecting applications using DSP/BIOS kernel services will enable you to build structured, yet flexible applications. Once you become familiar with the DSP/BIOS kernel and its capabilities, you will find developing applications easier to design, debug, maintain, and extend.

For experienced developers, the DSP/BIOS kernel implements most of the traditional kernel services found in embedded systems, and you will find DSP/BIOS kernel services efficient, scalable, and easy to implement.

This document will assist you in getting started using the DSP/BIOS kernel in your application design. Part of this document will convey the philosophy of the DSP/BIOS kernel, and the rest will go into examples of using the DSP/BIOS kernel to build applications. For complete and in-depth technical data on the DSP/BIOS APIs, please refer to the following documents: *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403), and *DSP/BIOS User's Guide* (SPRU303).

1.1 Why the DSP/BIOS Kernel?

The DSP/BIOS kernel is a feature-rich, yet scalable collection of kernel services that developers use to manage system-level resources and to build the infrastructure of DSP applications. These kernel services, tuned and optimized for size and performance, are available on TMS320C5000 and TMS320C6000 DSPs. If you think of the DSP as a processing resource, then the scheduling of this resource requires management. Managing system-level resources, including the DSP, is the role of the system software and of the DSP/BIOS kernel. In addition, the DSP/BIOS kernel makes it easier to transfer your application to another TMS320™ DSP by providing hardware abstraction.

The software required in typical embedded DSP systems is comprised of two general components: the application and the system software. See Figure 1. System software is responsible for managing the system resources for applications. System resources include the hardware devices on the target platform and the DSP. Above this layer is the application itself. It is important to distinguish the elements of a system design into these components. Your intellectual property is in the application. The system software provides the application with infrastructure.

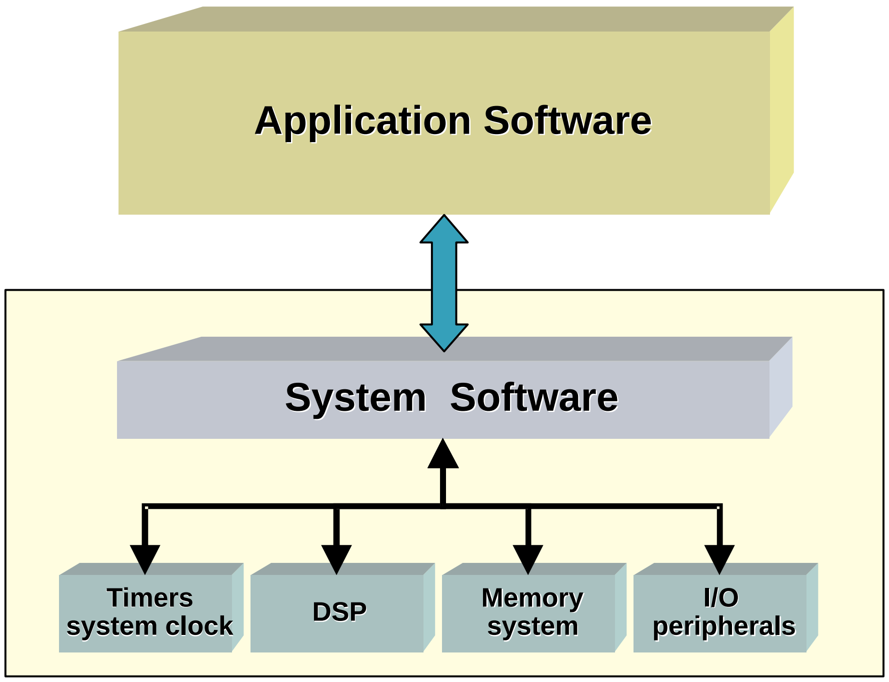


Figure 1. Embedded System Software Components

All DSP applications require some system software to manage the resources in their system. Typical embedded DSP systems include the DSP processor, memory, a system timer or clock, and I/O peripherals. System software is required to properly initialize and control these hardware components. In addition, the system software typically manages access between the system resources and the application software. This is important when transferring applications to different hardware platforms, such as the next generation device. The system software provides some isolation, or a layer between the application and the physical hardware.

TMS320 is a trademark of Texas Instruments.

In simple systems, the system software consists of basic hardware initialization, peripheral access functions, and hardware interrupt service routines (ISRs). Systems that are more complex require real-time scheduling of the DSP to ensure correct operation. Furthermore, as applications require concurrent access to hardware resources such as the DSP, memory, or I/O, the need for an efficient resource manager and scheduler becomes paramount. Managing these resources is precisely the benefit of using the DSP/BIOS kernel. The DSP/BIOS kernel provides system services to manage the DSP system hardware components and to provide applications with services that manage the DSP utilization. By using the DSP/BIOS kernel, you

- Manage the DSP MIPS efficiently using multithreading.
- Use standard interfaces for I/O and hardware interrupts.
- Define and configure system resources efficiently, such as system memory.
- Gain real-time visibility into the execution of your application using the real-time analysis tools.
- Add structure to your application, organizing it around a collection of inter-related threads.
- Make it easier to migrate to new TMS320 DSPs since DSP/BIOS abstracts much of the system hardware.

2 The Development Process

Since the DSP/BIOS kernel is a scalable set of componentized system services, you have complete control over which DSP/BIOS components you use. When you develop applications or port pre-existing applications to the DSP/BIOS kernel, you select only those components that your applications needs. Only those components you select are included in the build of your application. This keeps the memory usage by the DSP/BIOS kernel to a minimum.

Typical applications use the DSP/BIOS kernel to configure the system interrupt vector table and the system memory map. During the development process, most developers take advantage of the real-time analysis features built into the DSP/BIOS kernel to gain visibility into the run-time behavior of their application. In addition, most applications take advantage of the DSP/BIOS scheduler to prioritize and manage the DSP processing. Every application can take advantage of the features within the DSP/BIOS kernel. Pre-existing applications can take advantage of the DSP/BIOS kernel to make it easier to port to another supported TMS320 DSP, and take advantage of the real-time analysis features to better understand the runtime behavior and performance.

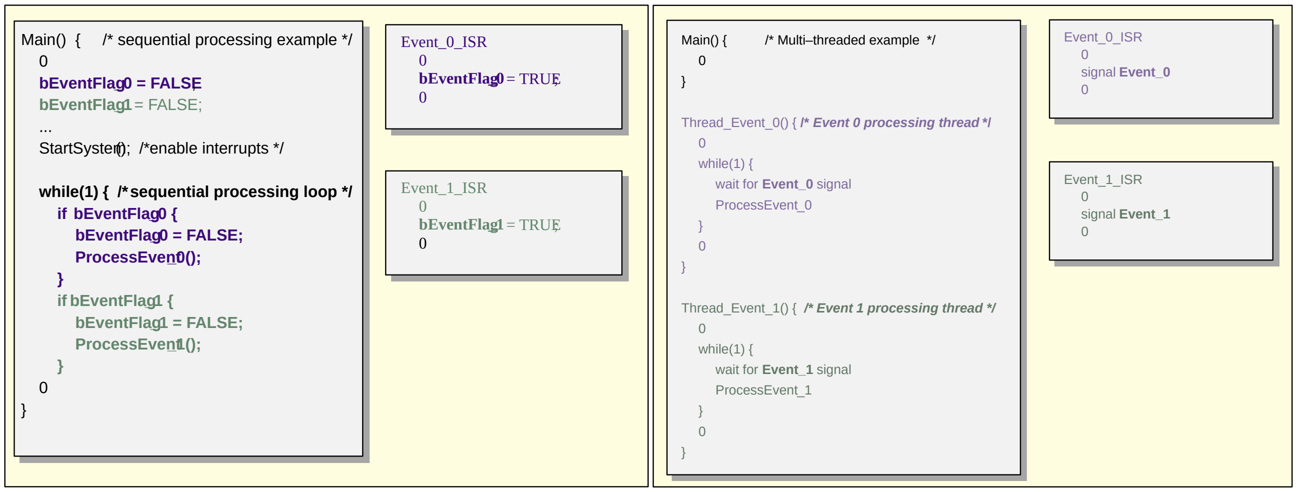
Concluding with the audio example that ships with Code Composer Studio™, is a description of the process of developing a DSP/BIOS application that uses several DSP/BIOS components. You will likely use one or more of these in your application. However, first the overall design process is described. While this process assumes you are developing a new application, many of the steps pertain to migrating your existing application to use the DSP/BIOS kernel as well. To help illustrate the process steps, a simple motor-control application is used.

2.1 Design With Multiple Threads in Mind

The overall process requires designing an application using a multi-threaded or concurrent system paradigm. While the DSP/BIOS kernel accommodates traditional loop architectures, multithreaded designs are far more flexible and maintainable. Even if your application consists of ISRs and a background operation, designing with DSP/BIOS threads will make it easier to accommodate new features or other changes over time including migrating to another TMS320 DSP.

Code Composer Studio is a trademark of Texas Instruments.

For many developers migrating to the DSP/BIOS process, the paradigm shift away from traditional processing loops or “super” loops to using multiple execution threads is new and perhaps challenging. However, organizing an application around a collection on inter-related threads provides structure to the application that makes it easier to maintain and extend over time. This structure allows developers to assign relative priorities to the threads to ensure the application executes correctly. Figure 2 illustrates a main processing loop restructured around threads.



```
Main()  {    /* sequential processing example */
   0
   bEventFlag0 = FALSE;
   bEventFlag1 = FALSE;
   ...
   StartSystem();  /*enable interrupts */

   while(1) {  /* sequential processing loop */
      if  bEventFlag0 {
         bEventFlag0 = FALSE;
         ProcessEvent_0();
      }
      if bEventFlag1 {
         bEventFlag1 = FALSE;
         ProcessEvent_1();
      }
   0
}
```

```
Event_0_ISR
   0
   bEventFlag0 = TRUE;
   0
```

```
Event_1_ISR
   0
   bEventFlag1 = TRUE;
   0
```

```
Main() {         /* Multi–threaded example  */
   0
}

Thread_Event_0() { /* Event 0 processing thread */
   0
   while(1) {
      wait for Event_0 signal
      ProcessEvent_0
   }
   0
}

Thread_Event_1() {  /* Event 1 processing thread */
   0
   while(1) {
      wait for Event_1 signal
      ProcessEvent_1
   }
   0
}
```

```
Event_0_ISR
   0
   signal Event_0
   0
```

```
Event_1_ISR
   0
   signal Event_1
   0
```

Figure 2. Traditional Main-Loop Processing (left) Restructured as a Multi-threaded Application (right)

Historically, DSP applications performed all the processing in an ISR. Over time, a background loop became necessary to perform some of the processing to minimize the amount of time spent in the ISR. Deferring some of the processing to the background reduced interrupt latency and increased interrupt bandwidth. As DSP application complexity increased, developers began to massage the background loops to perform rudimentary scheduling to implement state machines, or to ensure that higher-priority operations met their real-time deadlines.

These designs, however, still consisted of ISRs and a background-processing loop. As you can imagine, these designs are tricky with respect to timing, and are not very accommodating to changing requirements like adding new features. In most cases, these designs readily decompose into multiple independent paths of execution. With the DSP/BIOS kernel, you assign a thread for each of these independent execution paths. Since the threads must execute on the same processor, they are inter-related. Therefore, you will assign relative priorities to these threads to ensure the system performs correctly. DSP Bios uses a fixed-priority, preemptive scheduler that allows you to apply well defined real-time scheduling techniques such as rate monotonic scheduling (RMS) to ensure that your application runs correctly.

<u>Guideline</u>

In your design, you should minimize the ISR processing to be as little as possible, and defer the balance of the processing to a DSP/BIOS thread such as a software interrupt (SWI) or synchronous task (TSK). Doing this for all I/O allows you to prioritize each operation based on its importance or real time deadline.

2.1.1 DSP/BIOS Thread Types

The DSP/BIOS kernel supports four basic thread models. Hardware interrupt (HWI) and software interrupt (SWI) threads support fast interrupt processing. With the DSP/BIOS kernel, you need to rethink interrupt processing from single-level (hardware interrupts only) to a two-level model. The two-level model means you perform minimal processing in the hardware interrupt, and defer the rest of the processing to the software interrupt. Both HWI and SWI threads share the system stack; this makes their context switching times the shortest possible, making them ideal in time-critical operations.

The DSP/BIOS kernel provides a special software interrupt thread (PRD) to perform periodic functions. This thread executes as a function of a clock source. The clock source is either the system clock or a data clock. This thread is ideal for implementing periodic operations.

The synchronous task thread (TSK) processing model is the typical application execution thread. Tasks have the unique property of suspension, making them very flexible in operation. This flexibility makes tasks easy to use for a variety of applications. However, since tasks each have private stacks, their context switching times are longer than that of software interrupts. This suggests that the most time critical operations execute as software interrupts, and less time-critical operations use task threads.

The forth model is the background idle loop. This thread runs forever and loops continuously in the absence of any other thread ready to run.

2.1.2 Identify Independent Execution Paths

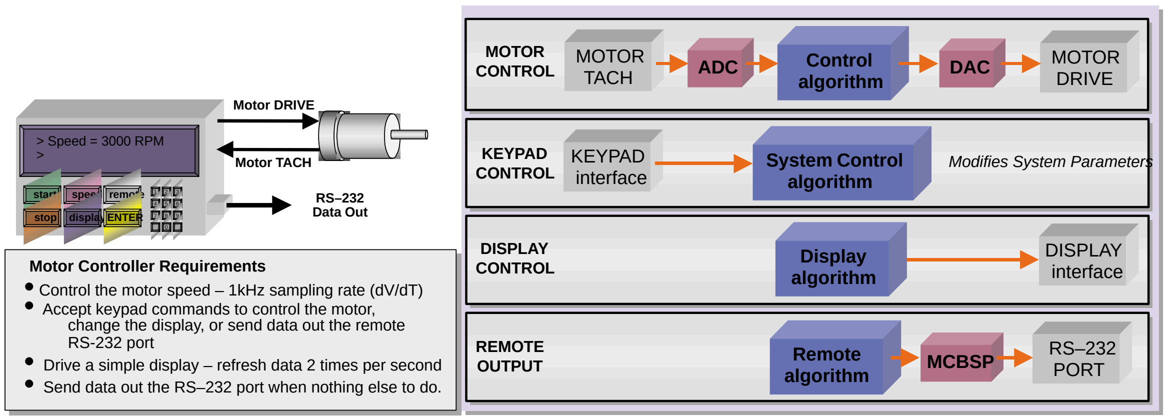
First, you need to architect your application using DSP/BIOS threads. To do this, you need to isolate independent paths of execution. Data flow diagrams, system-level block diagrams, and state machines provide good insight into both independent and dependent paths. These paths will end up as DSP/BIOS threads. Typical applications use multiple threads and thread types.

Guideline

When you map threads to the execution paths, choose the thread type that matches your run-time requirements. Which thread type to use is dependent on many factors, such as priority, latency, overhead, triggering, and dependencies on other threads.

2.1.2.1 Motor Control Application Example

Figure 3 illustrates a simple motor-control application. A single DSP is required to perform multiple functions: control the motor, respond to the keypad, drive a display device, and send data out the remote port. The user uses the keypad to control the system. For purposes of this example, the motor control algorithm is a simple speed control. A timer interrupt fires every millisecond (1 kHz rate), and triggers the motor control algorithm to read the current speed (TACH value) and adjust the motor drive to attain the desired motor speed. The user uses the keypad to control the system. Controlling the system includes controlling the motor, selecting what to display, and controlling what to send to the remote port. When enabled by the keypad selection, the remote port transmits diagnostic data.



Motor Controller Requirements

- Control the motor speed – 1kHz sampling rate (dV/dT)
- Accept keypad commands to control the motor, change the display, or send data out the remote RS-232 port
- Drive a simple display – refresh data 2 times per second
- Send data out the RS–232 port when nothing else to do.

Figure 3. Simple Motor Control Application Separated into Independent Execution Paths

Figure 3 also illustrates this application as independent execution paths. Each of the major functions executes independently. Although the keypad control algorithm modifies system parameters to control the other functions, the motor control operation for instance is not dependent on the execution of any of the other system functions.

2.1.3 Identify Real-Time Deadlines and Critical Operations

Real-time systems have real-time deadlines for operations to complete. The critical operations in your application will likely have real-time deadlines, and must execute at higher priority than your non-critical operations. You need to determine all real time deadlines in your system, and assign relative priorities to each. This information is necessary to map your operations to the appropriate DSP/BIOS execution threads. Your most critical threads will likely execute using high-priority SWI or TSK threads.

In addition to priority, thread latency may impact meeting your real-time deadlines too. Latency used in this context relates to the context switching times in activating a thread, and measures the time it takes to activate the thread once triggered. In the DSP/BIOS kernel, the hardware interrupt has the shortest latency, or restated the fastest context switching time. Following closely are the software interrupts. This is a result from both the hardware and the software interrupt using the same stack as the application. However, each DSP/BIOS synchronous task has a private stack. Consequently, the context switching time for tasks is longer than that for software interrupts.

Guideline

As a rule, for your most time critical threads, use either a hardware interrupt or a software interrupt thread. For less critical requirements, the task thread may be more convenient.

For more details and analysis on DSP/BIOS thread latencies and other DSP/BIOS objects, please refer to *Benchmarking DSP/BIOS on the C6000* ™ (SPRA662) and *Benchmarking DSP/BIOS on the TMS320C54x* ™ (SPRA 663).

C6000 and TMS320C54x are trademarks of Texas Instruments.

2.1.3.1 Motor Control Example

Controlling the motor is the most critical function in our example, so it becomes the highest priority operation in the system. Responding to the keypad is also important since this is the control input. However, the response time is not as critical as smooth motor operation, so it has a lower priority than the motor control operation. The display provides information and is not critical to the operation, so it is a lower priority than the keypad. Finally, sending data out the remote port is a background operation in our example, so it has the lowest priority.

2.1.4 Specify Thread Triggering and Synchronization

Each independent path of execution requires an activation mechanism. Hardware devices trigger an interrupt service routine managed by HWI. Software triggers DSP/BIOS SWI and TSK threads. The idle loop runs continuously in the background. In addition to triggering, your application may need to synchronize multiple trigger sources.

Each DSP/BIOS execution thread type uses unique activation and synchronization methods. The DSP/BIOS software interrupts (SWI) execute in a similar fashion to their hardware counterparts. Once triggered, they run to completion and terminate. These threads are re-triggerable. Software interrupt threads execute at a very high priority, just below hardware interrupts, and above synchronized tasks. To synchronize multiple trigger sources, the software interrupts use a special control interface called the SWI mailbox. The software interrupt will begin executing based on the contents of this mailbox. You would use this interface to perform conditional triggering on the software interrupt. Software interrupt threads are reactive. That is, activation is under external control. The thread cannot affect its own execution. Once started, it cannot wait for other events or stop before completing (suspend). As a rule, all input needed by a software interrupt’s function must be ready (available) before an event triggers the software interrupt.

Synchronized task (TSK) threads operate quite differently from interrupts. Tasks are not directly re-triggerable. You can use a while loop within the task function to act in a re-triggerable fashion. Task threads have the unique property of suspension, which allows them to affect their own execution. Tasks should suspend when they wait for resources, such as data from a peripheral or access to shared memory, rather than poll. Suspension also allows tasks to stop themselves by sleeping for some amount of time, or explicitly yielding the processor. Tasks pend on semaphores (suspend execution until ready) to synchronize execution. This gives you flexibility in the use of task threads. Using semaphores allows you to synchronize task execution to other events or other threads, or arbitrate access to system resources: typically, your normal execution paths, control functions, and non-critical I/O use tasks.

The idle loop executes in a similar fashion to traditional main loops. Each function in the loop executes in order, and runs to completion. You add your functions to the list using the Configuration Tool. The idle loop runs continuously in the absence of any higher-priority thread ready to run. The traditional ISR and background processing loop architectures can take advantage of the DSP/BIOS kernel by moving the traditional loop functions first into this idle loop. Then to improve performance or add functionality, you can move functionality to the other threads.

All DSP/BIOS threads are fully preemptable by higher-priority threads. In your design, you will need to identify the triggering or activation method for each execution thread, and determine if you want to execute portions of you application in the idle loop.

2.1.5 Identify Thread Lifetimes

The DSP/BIOS kernel allows you to create threads that will exist for the duration of the application. These are **static threads**. You also have the choice to instantiate SWI and TSK threads dynamically, such that they exist as long as the application needs them. Then, to free resources, you may delete them. These are **dynamic threads**. You create static threads using the Configuration Tool, and you create/delete dynamic threads using DSP/BIOS APIs in you application. The primary advantage of static threads is reduced memory requirements. Dynamic DSP/BIOS objects require more code to support the dynamic creation and deletion, and the creation process may block waiting for memory allocation to complete. However, you should know that once created, the execution time (and MIPS) is identical.

Static threads also support the real-time analysis tools better than dynamic threads. The execution graph does not have direct visibility into dynamically created threads, and cannot display as detailed information as it does with static threads. Dynamic threads, however, allow you greater flexibility in application designs.

You need to identify which threads in your application need to be static, and which threads you will instantiate dynamically.

Guideline

To minimize code space use static threads.

2.1.6 Identify Periodic or Multi-Rate Operations

Many DSP applications process data periodically. For example, audio data sampled and packed into 20 ms sized buffers requires processing each buffer every 20 ms. Multi-rate systems imply that multiple periodic operations will occur at different rates. The DSP/BIOS kernel provides a periodic function manager, PRD, that you will configure to execute your periodic functions. See Figure 4, which illustrates the periodic thread operation. Recall from *DSP/BIOS Technical Overview* (SPRA646), that periodic functions execute as a special SWI called the PRD_swi.

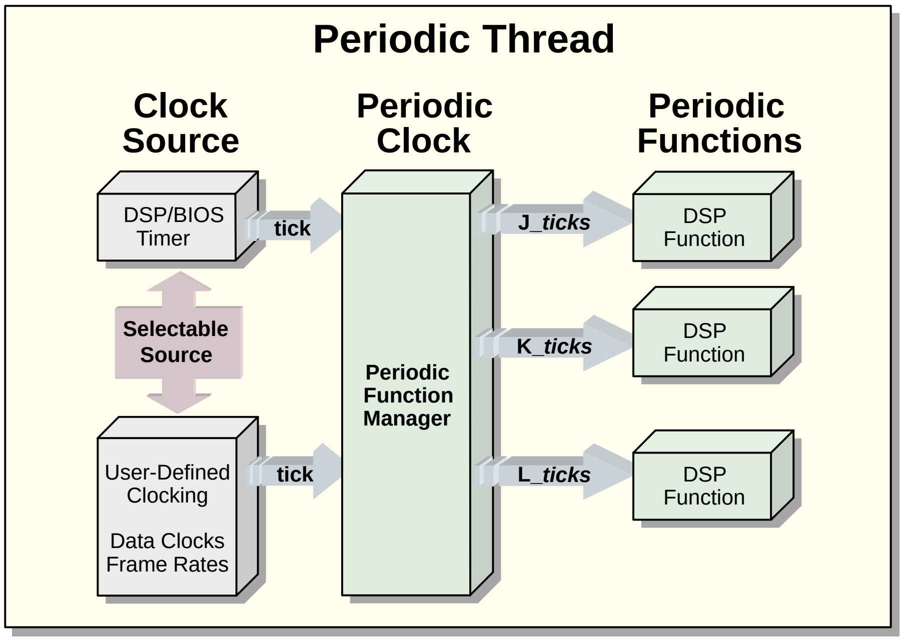


Figure 4. DSP/BIOS Periodic Thread Executes All Periodic Functions

Guideline

You need to identify all the periodic operations in your application. For all periodic operations in your application, determine the period of the operations that need to run, and the time to complete the operation once it has been initiated. All of your periodic functions need to occur at integer multiples of the PRD tick. Therefore, you need to make sure you set the PRD tick period appropriately. To prioritize the periodic functions, you should have the PRD functions post software interrupts or synchronized tasks to perform the operation, which you prioritize as required See Figure 5.

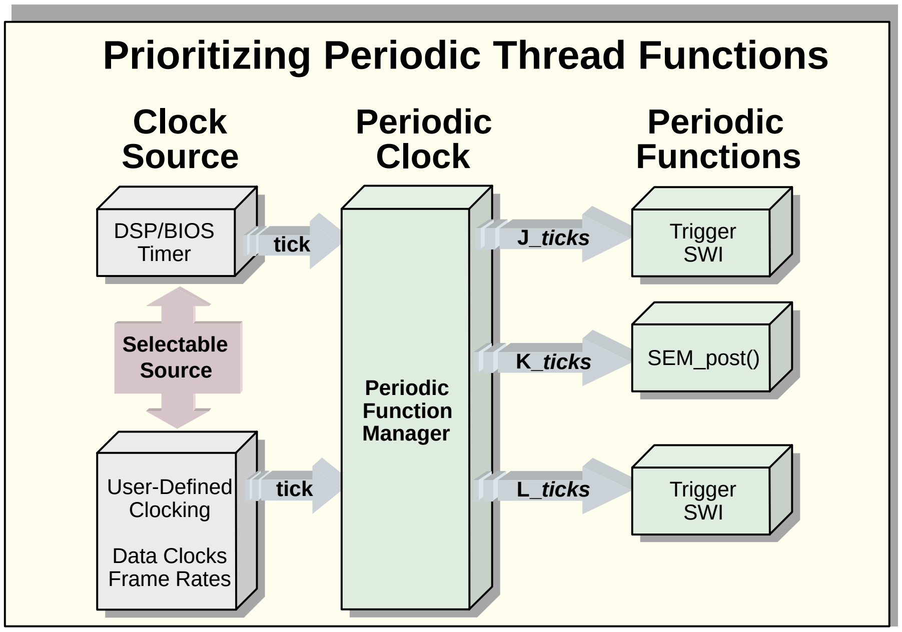


Figure 5. Prioritizing Periodic Functions

For example, an audio algorithm may require processing 20 ms size buffers and take 5 ms to execute. This algorithm operates on continuously streaming data. If a buffer is dropped (not ready for output), then the output will contain distortion sounding like pops. The application must ensure that the algorithm completely executes (all 5 ms), within the 20 ms window (buffer period), otherwise you will drop a buffer. The period is 20 ms, and the real-time deadline is 20 ms, since you do not want to drop a buffer.

Some applications perform periodic operations on data from sources that supply their own clocking. For example, frames of data transferred by the DMA, perhaps originating from a serial port (MCBSP). The frame rates drive the clocking. If your application needs to perform periodic operations on this type of data, you will want to drive the PRD clock from the interrupt service routine that handles this data. In this way, you can perform periodic operations at the tick or frame rate of the incoming data. In the previous example, the DMA interrupt would call `PRD_tick()` to advance the PRD clock after each frame transfer.

The default setup uses the DSP/BIOS system clock to drive the PRD clock, and is set-up through the CLK manager in the Configuration Tool. This allows periodic functions to execute at integer multiples of the system clock.

2.2 Identify Data Paths and Buffer Requirements Throughout the Design

In many DSP applications, the data flow from input to output is often a continuous flow of data blocks or buffers. This continuous flow of data, referred to as data streaming, is typical of audio, video, and speech applications. Other data flows may be non-continuous, such as messaging and communication paths. The DSP/BIOS data pipes (PIP) and data streams (SIO) are well suited to manage streaming data. In your application, you will use these objects to pass buffers between I/O devices and your application, or between DSP/BIOS threads within your application. In your design, you need to isolate data paths that involve more than one thread. To send streaming data between threads, you use either PIP or SIO.

Streaming data applications require management of the flow of data buffers throughout the application. Typically, DMA devices move data from peripherals to system memory, or the opposite path, system memory to the peripheral. The DMA transfer-complete interrupt signals the application that the data is available (for data input), or memory is available to output data. However, once in the application, managing buffer movement is not as straightforward. Application designers devise their own non-standard methods to accomplish this. DSP/BIOS data pipes and data streams are kernel objects optimally designed to perform these common tasks.

The PIP module provides a minimal architecture that is code-efficient, fast, and flexible. You can think of PIP as a *software*-DMA that performs the transfer of data between two DSP/BIOS threads, and provides both event-driven and polling interfaces. Much as a DMA device issues an interrupt when a block transfer is complete. PIP can execute a callback function to alert the target thread that a block is available.

Figure 6 illustrates a typical mapping of data paths in a block diagram, and the implementation using data pipes (PIP).

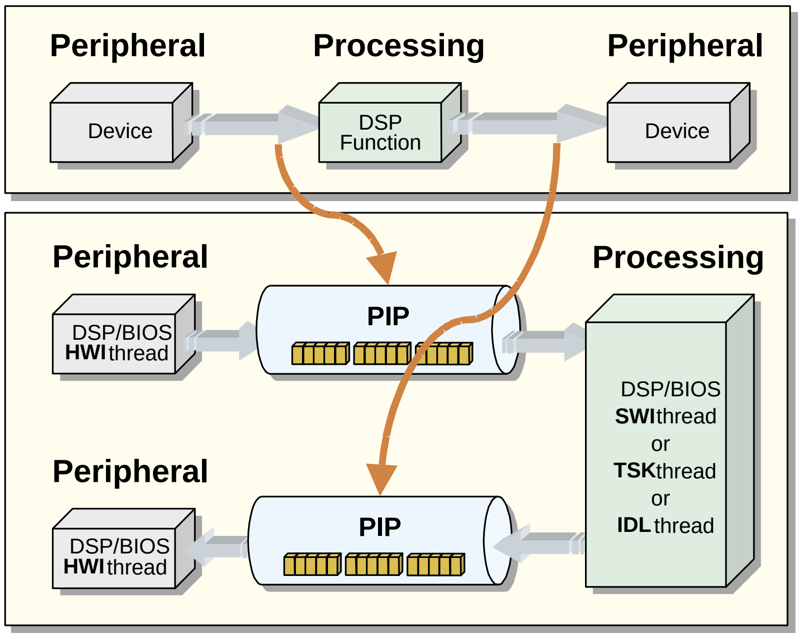


Figure 6. Generic Mapping of PIP to Block Diagram Data Paths

Alternatively, you can use the SIO module. The SIO module provides a high-level mechanism to perform streaming I/O between a synchronous task thread and a device. SIO provides a simple common interface independent of the underlying device. This level of abstraction allows application designers to easily communicate with any device.

SIO provides APIs to create and delete I/O channels, control the data stream such as start, stop, idle and flush; and to transfer data in the channel, get, put, issue and reclaim. SIO supports static creation as well as dynamic creation and deletion.

Figure 7 illustrates a typical mapping of data paths in a block diagram and the implementation using data streams (SIO). Note that SIO links device drivers to synchronous tasks. They are referred to here as drivers rather than devices, since device drivers provide the isolation from SIO to a device or another thread. In fact, using the DPI device driver supplied with the DSP/BIOS kernel allows you to stream data between two or more synchronous tasks.

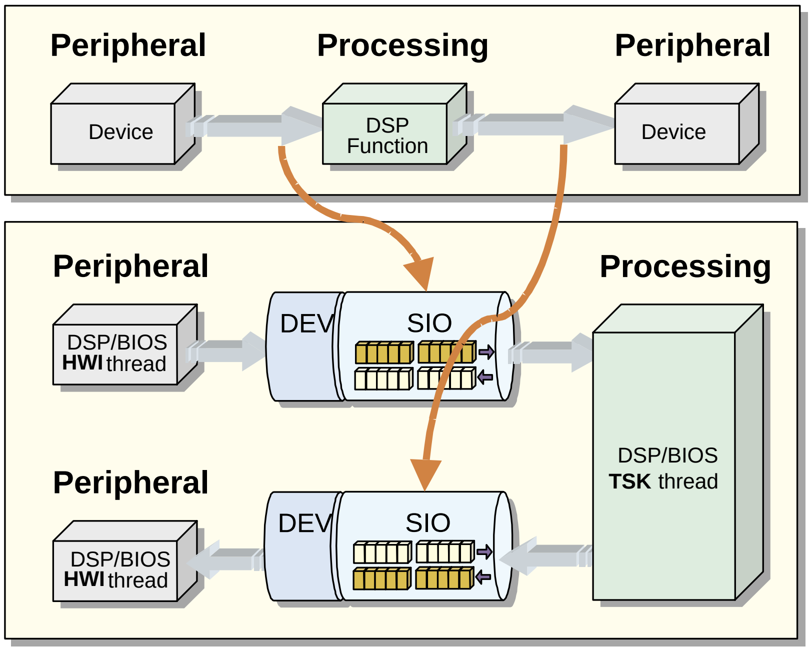


Figure 7. Generic Mapping of SIO to Block Diagram Data Paths

Both PIP and SIO are flexible, since the content of the buffers is user-defined. The buffers can contain data directly, addresses to data buffers, or anything else. It is important to note that both PIP and SIO pass pointers to buffers between threads; they do not copy the contents. This makes the size and the contents of the buffers independent of the transfer. This allows the transfer time to be constant, a must for real-time systems.

To understand how to architect multichannel applications using PIP and SIO, please refer to *Using PIP and SIO in Multichannel Systems* (SPRA689)**.** To understand how to write device drivers that will interface with PIP and SIO, please refer to *Writing Flexible Device Drivers for DSP/BIOS* (SPRA700).

2.2.1 Identify Inter-Thread Communication and Messaging Paths

It is common for threads to communicate with one another. This is typical of control systems that send messages to tasks to command their operations. The DSP/BIOS kernel provides mailboxes to send messages (or anything else defined in the message being sent) between threads. Mailboxes differ from PIP and SIO, since messages posted to the mailbox are copied into the mailbox. You would use these objects to communicate messages typically between threads. For example, you create a supervisor task that receives and transmits messages to other processing tasks in the system.

As with PIP and SIO, the contents of the messages are not restricted. You can pass anything in the message such as commands, raw data, and pointers to memory buffers.

Queues in the DSP/BIOS kernel provide the simplest mechanism to communicate between any two threads without copying. DSP/BIOS queues are double-linked lists that threads insert and remove elements from. You can think of queues as *software*-FIFOs. Queues are not limited to communication between threads; however, this is their most common use. In fact, SIO uses two internal queues to pass buffers between the task and the device driver on the same task thread.

2.3 Identify the DSP/BIOS Objects You Will Use

Once you have a multi-threaded design, you need to identify the DSP/BIOS objects you will use in the application. From the design, you will map the appropriate thread type to each execution path. If there are dependencies, you need to determine the interaction and synchronization required.

You also need to identify the data paths and the characteristics of each. This includes buffer sizes and formats. For I/O device communication, select either data pipes (PIP) or data streams (SIO). For inter-thread communication, you can use data pipes or stream, data queues (QUE), or mailboxes (MBX).

2.4 Use the Configuration Tool to Pre-Configure the System and DSP/BIOS Objects

The DSP/BIOS kernel needs to be aware of several global system parameters. These include the DSP device, the CPU clock speed, endian mode, cache settings, and some others. Use the DSP/BIOS Configuration Tool in CCStudio to pre-configure your target system settings.

- You use the Configuration Tool to select and configure the DSP/BIOS run-time support objects needed for your application. Using these objects, you develop and validate the logic of your application by building an application framework that represents the execution threads, I/O, and their interactions.
- For DSP/BIOS threads, you specify the thread priority and the function the thread will call when activated. For software interrupts (SWI), you will also specify two function arguments that will pass with the function when the thread is activated; and you will specify the initial SWI mailbox value. For synchronized tasks (TSK), you will specify the function arguments that will pass with the function when the thread activates; and you will specify both the task stack size and the memory segment that the task stack will occupy. For idle functions, which execute in the background idle loop, you will specify the function to call.
- You use the Configuration Tool to create the system memory map. In non-DSP/BIOS designs, you would specify this map in the linker command file under the MEMORY directive. You also use the Configuration Tool to specify the sections the DSP/BIOS kernel will occupy and use; and to specify the system stack size and section. You may optionally specify the C compiler sections here, too.
- You use the Configuration Tool to create the interrupt vector table. In non-DSP/BIOS designs, you would specify this table in a separate file, usually in Assembly.
- You use the Configuration Tool to program the on-chip timer to operate as the DSP/BIOS system clock. This system clock is required for the real-time analysis features and sources other than DSP/BIOS time functions. The visual interface allows you to specify the timer interrupt rate in units of microsections per interrupt, and the Configuration Tool generates the required register settings.
- You use the Configuration Tool to program the peripherals using the Chip Support Library. Configure the Direct Memory Access (DMA) the Multichannel Buffered Serial Port (McBSP) and other peripherals from the Configuration Tool.

Figure 8 summarizes what you will do using the Configuration Tool to configure the DSP settings, and to pre-configure the static DSP/BIOS objects you will use in your application.

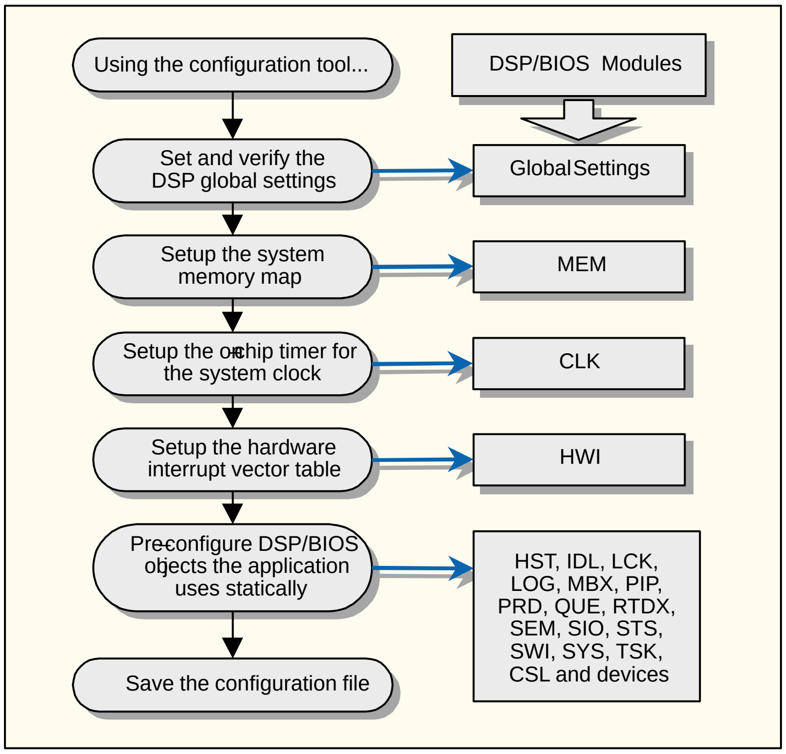

Using the configuration tool...
DSP/BIOS Modules
Set and verify the DSP global settings
Global Settings
Setup the system memory map
MEM
Setup the on-chip timer for the system clock
CLK
Setup the hardware interrupt vector table
HWI
Pre-configure DSP/BIOS objects the application uses statically
HST, IDL, LCK, LOG, MBX, PIP, PRD, QUE, RTDX, SEM, SIO, STS, SWI, SYS, TSK, CSL and devices
Save the configuration file

Figure 8. Using the DSP/BIOS Configuration Tool

2.5 Call DSP/BIOS APIs From Your Application

Once the static configuration is complete, you will make DSP/BIOS API calls from within your program to access and manipulate DSP/BIOS objects.

The DSP/BIOS kernel allows you to use the application's `main()` function to perform application initialization. You can think of the `main()` function as a create phase, in which you allocate memory buffers and do other initializations that need to be set and available before your application actually starts.

After `main()` is called, the DSP/BIOS kernel globally enables interrupts and starts running.

3 Code Composer Studio Audio Example

This section describes the Audio example that ships with Code Composer Studio. This is a simple application that illustrates the use of HWI, SWI and PRD threads. It also illustrates the use of data pipes (PIP objects) to manage the flow of data throughout the application.

The first step in this example is to create the block diagram of the intended application. See Figure 9.

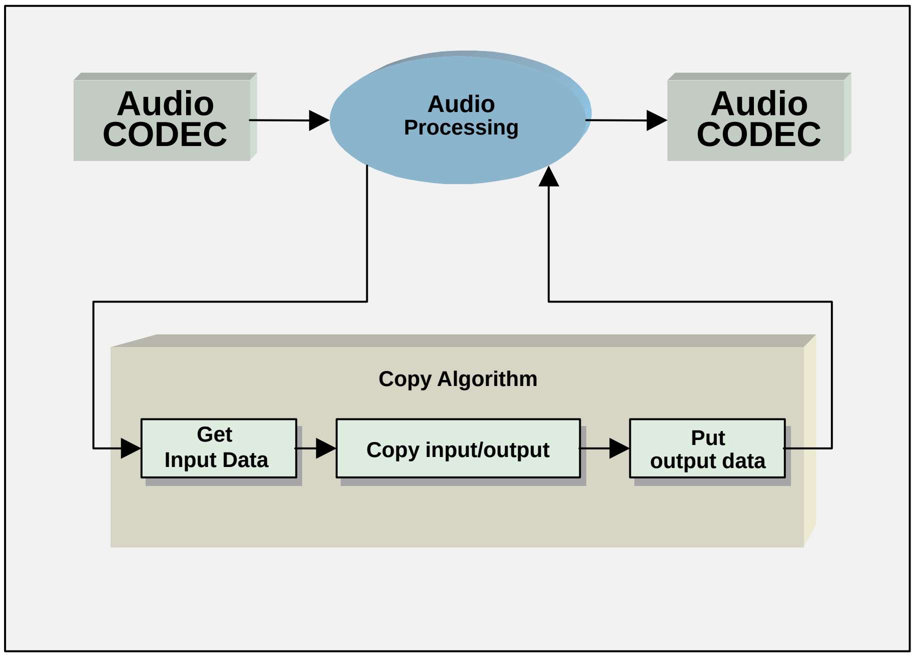


Figure 9. Audio Processing Requirement Functional Block Diagram

The application needs to acquire audio data from the CODEC, perform some processing, and issue the data out the CODEC again. For simplicity, our example just copies the audio input to the output. Figure 10 illustrates this in the part labeled *Processing Threads*.

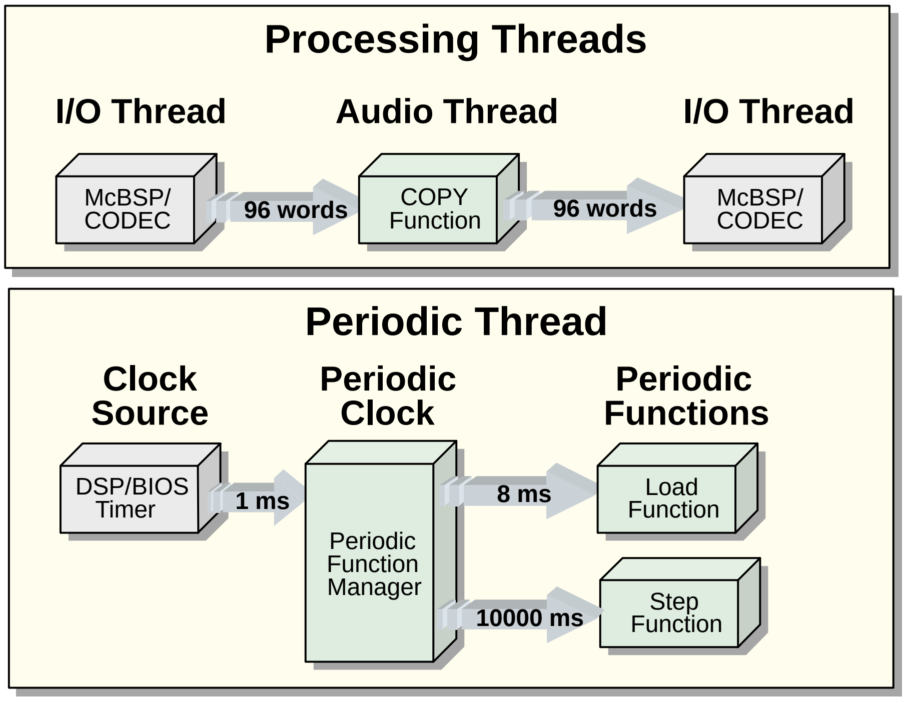


Figure 10. Audio Example Block Diagram

In addition to the audio processing, you want to periodically load the CPU to view the impact on your audio processing. This loading is incrementally increased and decreased to stress the application. Figure 10 illustrates this in the section labeled *Periodic Thread*.

Now that you have the block diagram that isolates the execution paths, you can assign DSP/BIOS threads. Since the audio-processing path is very simple, you only need to use one thread to do the processing. The design is simplified by keeping the sample rates at the input and output identical; therefore, the same ISR is used to handle input and output. HWI is used to manage the interrupt. Either a SWI or a TSK tread could be used to do the audio processing; however, the SWI is used in this case just to illustrate the use of the SWI mailbox to synchronize multiple trigger sources. With a TSK thread using the SIO module, this synchronization is not required.

You want the SWI to do the audio processing only when there is a full buffer of input data, and you have an available empty buffer to store the processed data. To manage the flow of these buffers, data pipes are used to illustrate the callback operation that will trigger the SWI audio processing. Figure 11 illustrates the mapping of these DSP/BIOS objects onto the audio processing threads. Thus, in this simple example, there is one HWI thread handling the CODEC interrupts, and one SWI thread is used to handle the audio processing. The interrupt processing is one execution path, and the audio processing is a separate execution path. Connecting these execution paths or threads, are data pipes, which manage the buffer flow.

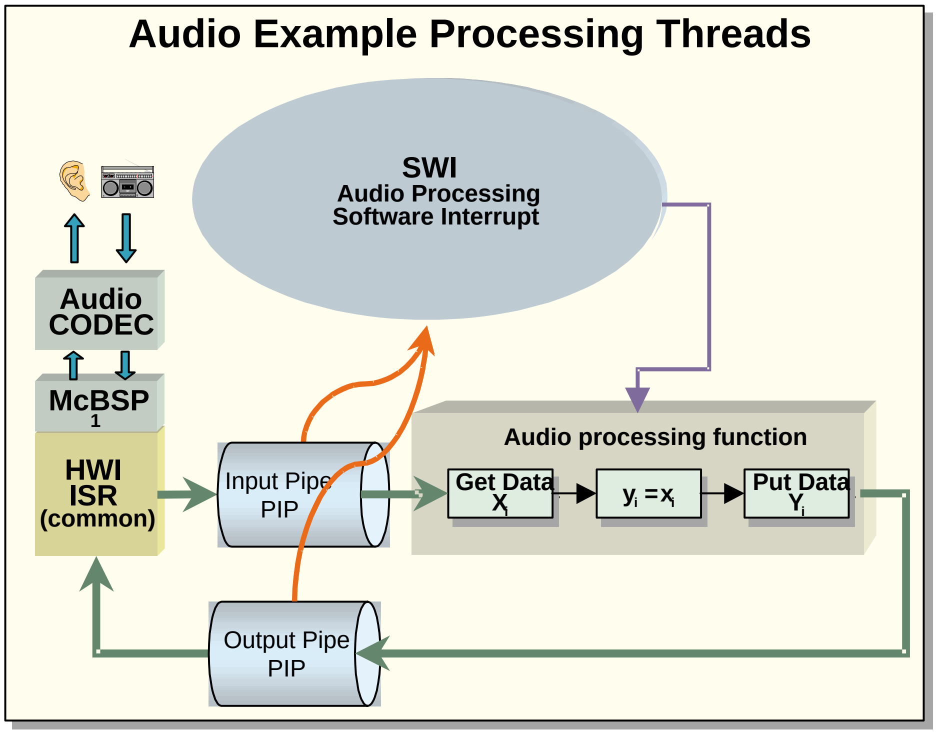


Figure 11. Mapping the Block Diagram to DSP/BIOS Threads

Independent of the audio processing, our example has periodic functions to add CPU loading. The system clock is being used to drive the PRD clock with one-millisecond ticks. In this example, the PRD functions execute directly in the `PRD_swi` thread. The load function occurs every 8 milliseconds, and the step function executes every 10,000 milliseconds, or 10 seconds. The step function modifies the load function by increasing and decreasing the load period (see Figure 10).

For the example, the hardware implementations built on the TMS320C6201 EVM, TMS320C6211 DSK, TMS320C6711 DSK, and the TMS320C5402 DSK are referenced. See Figure 12. The multichannel buffered serial port, McBSP1, connects to the audio CODEC. This example uses an ISR (`DSS_isr()`) to interact with the MCBSP to read and write audio data, and uses the DSP/BIOS HWI module to manage hardware interrupts. The CODEC ISR fills empty blocks with input data, and outputs full blocks until they are empty.

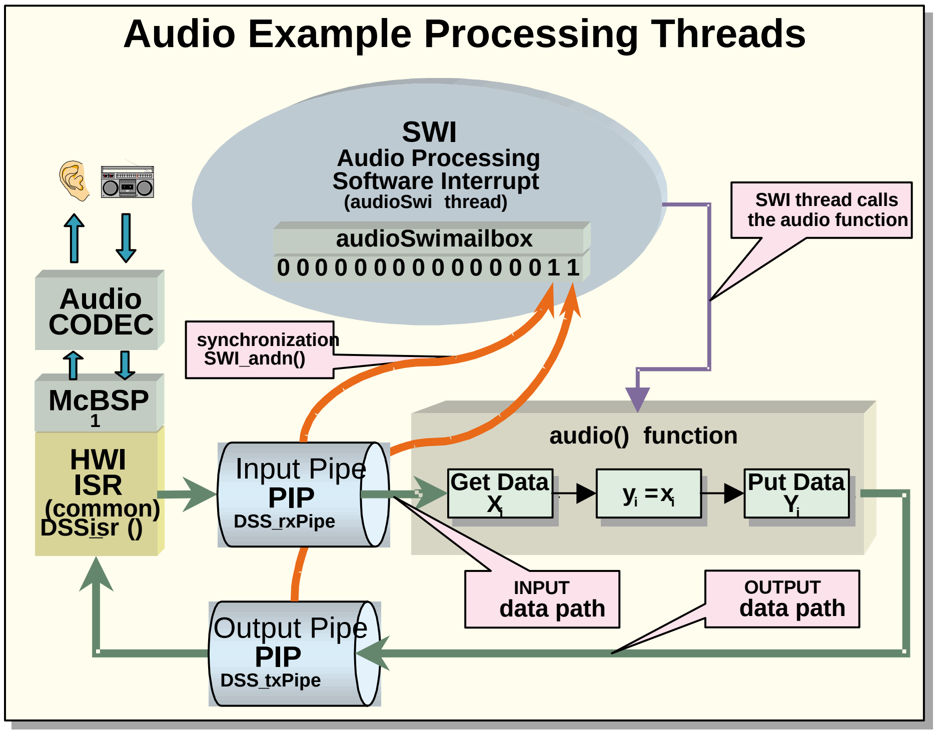


Figure 12. Audio Processing Operation Using the DSP/BIOS Kernel

To transfer data between the ISR and the application, DSP/BIOS data pipes managed by the PIP module are used. One data pipe transfers data from the ISR to the application (`DSS_rxPipe`); the other transfers full data to the ISR for output (`DSS_txPipe`).

The `audio()` function attached to the DSP/BIOS software interrupt thread (`audioSWI`) performs the audio processing. The audio processing thread activates only when both a full block of data and an empty block of data are available. To synchronize these events, the `audioSWI`'s mailbox is used. The initial value of the SWI mailbox is set to 3, which sets the first 2 bits in the SWI mailbox to 1's. When both of these bits become 0 (value of the mailbox is 0), the SWI thread activates to perform the processing. You will use the data pipe notify functions to clear these bits individually.

Both data pipes signal the software interrupt using `SWI_andn()` calls to their assigned bits in the SWI mailbox to synchronize the processing. The input data pipe will signal the `audioSWI` when the ISR has filled a block, and it is available for processing by calling `SWI_andn(2)` to clear bit 1 in the SWI mailbox. Likewise, the output data pipe will signal the `audioSWI` when an empty block of data is available for the application to fill by calling `SWI_andn(1)` to clear bit 0 in the SWI mailbox.

Figure 13 illustrates the periodic thread configuration. Note that the system clock is used to drive the periodic clock, and the tick period is one millisecond. The PRD manager will call the `load()` every 8 milliseconds, and the `step()` function every 10 seconds.

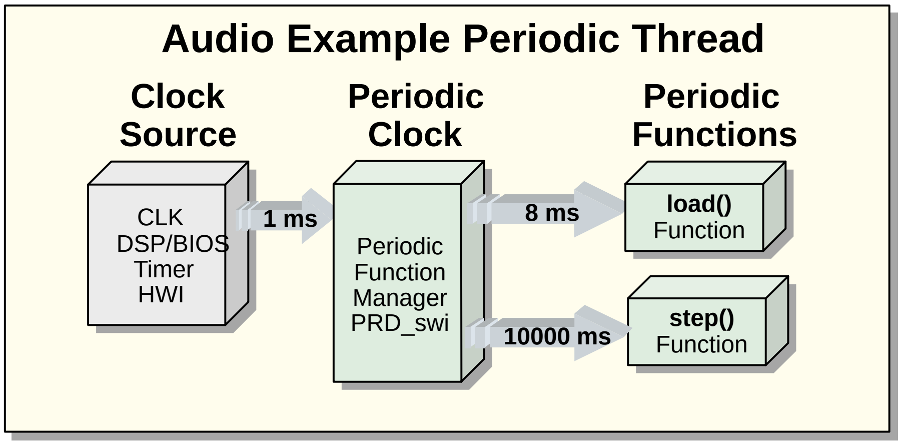


Figure 13. Load Processing Operation Using the DSP/BIOS Kernel

All of the DSP/BIOS objects used in this example are static. The audio.cdb file contains their configuration information. Within the application, you can observe the API's calls used to access buffers in the data pipes, both in the ISR and in the `audio()`.

4 Conclusion

Understanding how to build applications using the DSP/BIOS kernel requires adopting a multi-threaded design paradigm. However, once understood, designing DSP/BIOS applications becomes straightforward and you will find developing applications easier to design, debug, maintain, and extend. As you can see from the motor control and audio examples, decomposing an application into independent execution paths is not as difficult as it first appears. Often, reviewing system-level block diagrams yields insight into the separation.

After you have divided the application into independent threads, you need to assign the appropriate DSP/BIOS thread type. the DSP/BIOS kernel provides four basic thread types that allow you to optimally apply, based on the execution requirements.

You then choose the appropriate data flow objects to pass data throughout your application. The DSP/BIOS kernel offers device-independent I/O objects to handle continuous block data transfers. For inter-thread communication and other transfers, the DSP/BIOS kernel provides mailboxes and data queues.

To build your application, the DSP/BIOS kernel provides a system Configuration Tool to specify and configure system components, such as the system memory map and hardware interrupt vector table. You also use this tool to statically declare the DSP/BIOS objects (threads and I/O) that you will use in your application.

This provides the foundation and infrastructure that your application will use. At this point, you build your application using these components and services.

5 References

1. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403).
2. *DSP/BIOS User's Guide* (SPRU303).
3. *DSP/BIOS Technical Overview* (SPRA646).
4. *Benchmarking DSP/BIOS on the C6000* (SPRA662).
5. *Benchmarking DSP/BIOS on the TMS320C54x* (SPRA663).
6. *Using PIP and SIO in Multichannel Systems* (SPRA689).
7. *Writing Flexible Device Drivers for DSP/BIOS* (SPRA700).