Application Report
SPRA646 - March 2000

DSP/BIOS II Technical Overview

David Dart *Software Development Systems*

ABSTRACT

DSP/BIOS II is a kernel that provides run-time services which developers use to build DSP applications and manage application resources. DSP/BIOS II effectively extends the DSP instruction set with real-time, run-time kernel services that form the underlying architecture, or infrastructure, of real-time DSP applications.

The DSP/BIOS II kernel tightly integrates with the Code Composer Studio™ Integrated Developers Environment (IDE) to provide the ability to:

- select and configure the foundation modules and kernel objects required by the application with the DSP/BIOS Configuration Tool;
- provide DSP/BIOS II kernel object viewing with the Code Composer Studio (CCStudio) plug-in utility; and,
- support the real-time analysis features in DSP/BIOS II with host-side tooling.

Contents

List of Figures

1 DSP/BIOS II in the Application Development Cycle

Where does DSP/BIOS II intersect the application development process? DSP/BIOS II provides value from application design through deployment. DSP/BIOS II also provides the stable foundation upon which developers build and deploy their applications.

Application design cycles begin in the concept phase where developers propose and evaluate possible solutions to product requirements. Prototypes are typically developed and evaluated using a TMS320 EVM or DSK long before the target platform is available. At this phase, it is critical to build these prototypes quickly to ascertain and resolve technical uncertainties. To facilitate rapid application development, designers can leverage the DSP/BIOS II kernel and target hardware abstraction services to quickly build and test logical models of their applications.

Developers use the DSP/BIOS II Configuration Tool in CCStudio to select and configure the runtime support elements needed for their application from the scalable DSP/BIOS II kernel. Using these elements, designers develop and validate the logic of their application by building an application framework that represents the execution threads, I/O, and their interactions. Developers can also create a logical map of the target system memory to simplify the migration to the target platform(s) when it becomes available.

Since the DSP/BIOS II kernel contains components that implement a variety of execution thread models and device-independent I/O, designers can develop applications ranging from simple single-channel signal processing systems to very sophisticated multirate and multichannel systems. The Kernel Object View and Real-Time Analysis (RTA) features in DSP/BIOS II and CCStudio enable designers to quickly validate their application execution logic — long before they attach the algorithms that process the data. The DSP/BIOS II kernel objects created by the Configuration Tool contain internal instrumentation for real-time analysis. By using the real-time analysis utilities, designers can quickly measure the overhead associated with the application framework built using the DSP/BIOS II components. Once the designers validate the application's logic, they can add the algorithms. The RTA components in DSP/BIOS II also provide visibility into the run-time operation of the algorithms.

Developers can instrument their own algorithms with custom test vectors that support error detection and notification, data stimulation, and real-time data capture. Coupled with the Real-Time Data Exchange (RTDX) features, developers can also tune their algorithms by updating parameters and monitoring the result while the application is running on the DSP in real time.

During the final integration, errors or "glitches" often occur as a result of real-time interaction. These are hard to find since they typically are non-periodic or occur at a very low frequency. However, the RTA features built into the DSP/BIOS II kernel, coupled with any custom test vectors designed-in by the developer, provide unique visibility into the events that lead up to, or follow, the error. This visibility can greatly assist developers in isolating and fixing the difficult integration problems.

2 A Real-Time DSP Environment

The application development environment for today's mainstream TMS320 DSPs revolves around two principal categories of software tools, both integral to CCStudio:

– a program generation suite featuring a C compiler, assembler, and linker for the target DSP architecture; and

– a program debug facility for executing application software on a target DSP platform, linked to and controlled by the host via JTAG emulation hardware.

These tools are comparable in capability to their microcontroller counterparts and are essential to any software development environment. However, they fail to directly address a critical dimension of embedded systems in general and signal processing applications in particular — the dimension of real time.

DSP/BIOS II and CCStudio work together to extend the mainstream DSP development environment with an equally essential set of real-time software capabilities. Figure 1 depicts the junctures at which DSP/BIOS II and its attendant host utilities dovetail with CCStudio IDE.

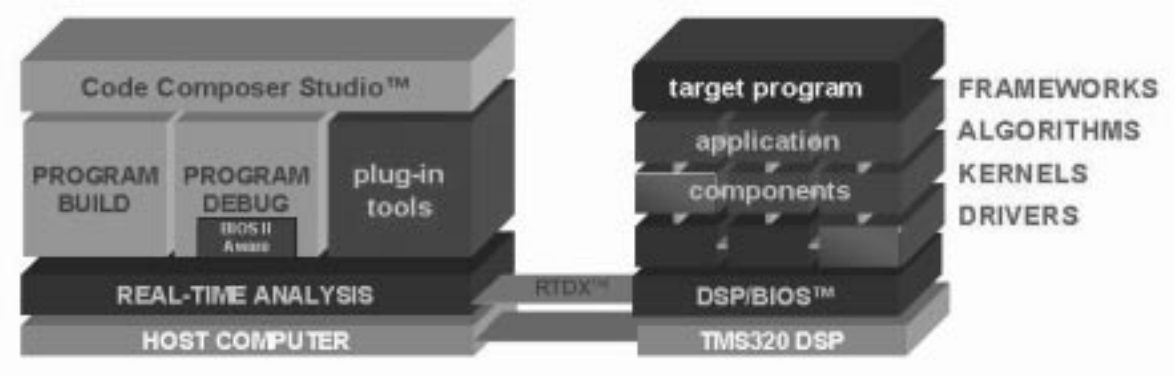

Figure 1. CCStudio IDE Enhanced with DSP/BIOS II

2.1 Developing DSP/BIOS II Applications

DSP/BIOS II is a scalable set of run-time services that provide the essential foundation upon which developers build their applications. DSP/BIOS II provides these run-time services in the form of a scalable run-time library, also referred to as the DSP/BIOS II kernel. Included in this library of services is:

- a small, preemptive scheduler for real-time program threads with optional multitasking support;
- hardware abstraction of on-chip timers and hardware interrupts;
- device-independent I/O modules for managing real-time data streams; and,
- a series of functions that perform real-time capture of information generated by the DSP application program during its course of execution.

Figure 2 illustrates the components involved in building DSP/BIOS II-based DSP applications.

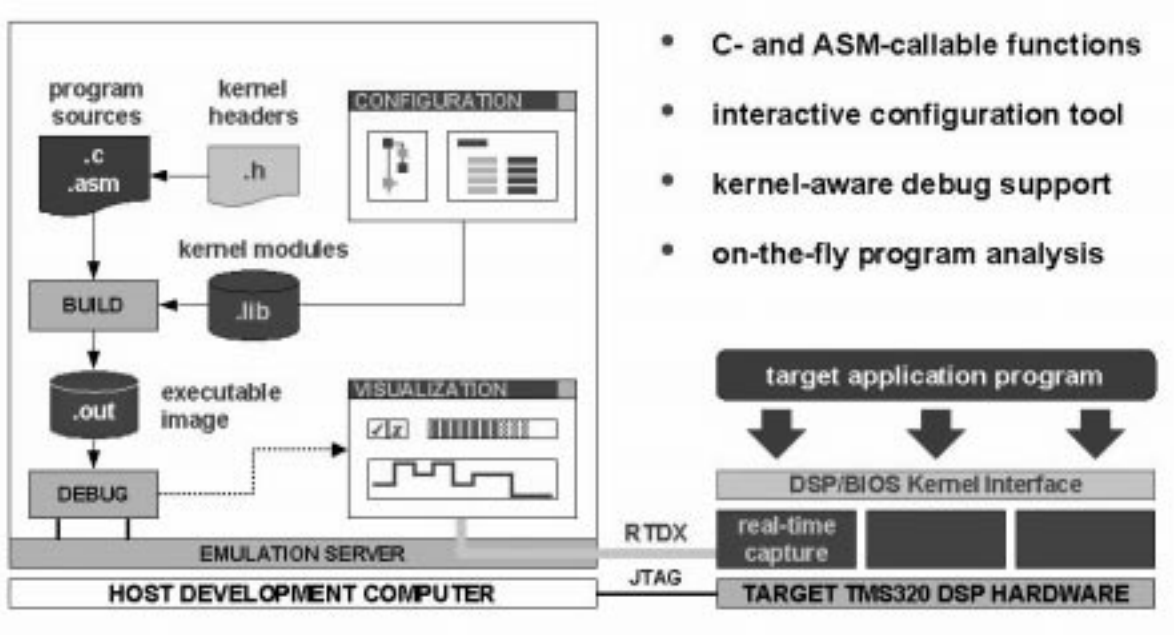

Figure 2. Building DSP/BIOS II-based DSP Applications

The target application invokes DSP/BIOS II run-time services by embedding corresponding API calls within its program source code.

DSP/BIOS II is essentially a library of functions callable from C (or assembly) programs. Developers prepare their source files, which include both DSP/BIOS II header files and API calls, in CCStudio. Programs then build (compile/assemble) in the usual fashion.

Developers use the DSP/BIOS II configuration tool to select which DSP/BIOS II modules they will use in their application; they may declare and configure objects from these modules as well. This pre-creates DSP/BIOS II objects to save memory and pre-validates the configuration parameters. The configuration tool generates four output files:

- *program*`cfg.s`***xx*** DSP/BIOS II Assembly source file. This file is linked with the application.
- *program*`cfg.hxx` DSP/BIOS II Assembly header file. This header file is included by the programcfg.s62 file.
- *program*`cfg.cmd` Linker command file. Used when linking the executable file. This file defines DSP/BIOS-specific link options and object names and generic data sections for DSP programs (e.g., .text, .bss, .data, etc.).
- *program*`.cdb` Stores configuration settings. This file is created by the Configuration Tool and used by both the Configuration Tool and the DSP/BIOS plug-ins.

For the C programmer, special header files included during program compilation define the DSP/BIOS II APIs within the context of the language. Alternative versions of many of the APIs support direct use of DSP/BIOS II services from assembly language programs using an optimized set of macros.

After compilation or assembly, the standard TMS320 linker binds the selected DSP/BIOS II services with the application into an executable program image (see Figure 2). The selected modules bind with the target application program, and may be located anywhere within the memory space of the target DSP platform, either in RAM or in ROM. The Configuration Tool provides the interface to locate the memory sections.

2.1.1 DSP/BIOS II Configuration Tool

The DSP/BIOS II Configuration Tool tightly integrates with CCStudio. This tool enables developers to select and deselect kernel modules, and control a wide range of configurable parameters accessed by the DSP/BIOS II kernel at run-time as shown in Figure 3. A file of data tables generated by the tool ultimately becomes an input to the program linker.

The DSP/BIOS II Configuration Tool (see Figure 3) serves as a special-purpose visual editor for creating and assigning attributes to individual run-time kernel objects (threads, streams, etc.) used by the target application program in conjunction with DSP/BIOS II API calls. The Configuration Tool provides developers the ability to statically declare and configure DSP/BIOS II kernel objects during program development rather than during program execution. Declaring these kernel objects through the Configuration Tool produces static objects which exist for the duration of the program. DSP/BIOS II also allows dynamic creation and deletion for many of the kernel objects during program execution. However, dynamically created objects require additional code to support the dynamic operations. Statically declared objects minimize memory footprint since they do not include the additional create code.

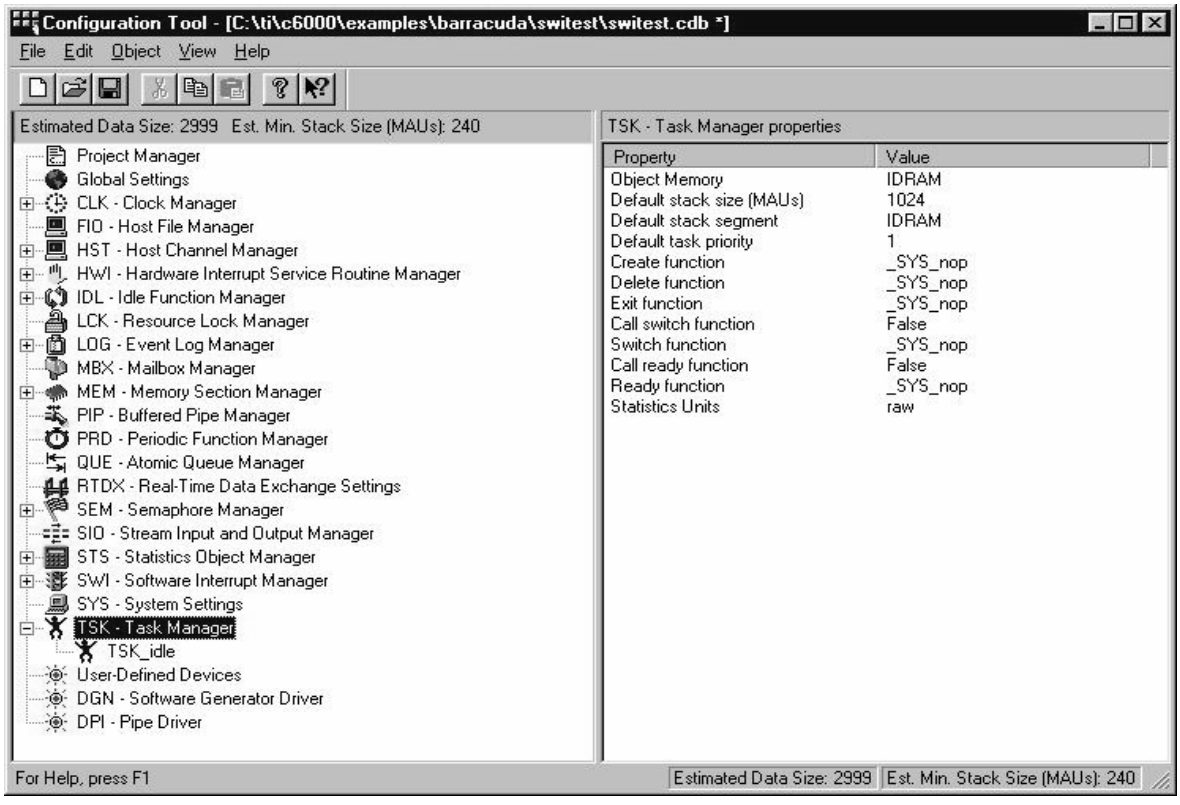

Figure 3. DSP/BIOS II Configuration Tool

Another important benefit of static configuration is the potential for static program analysis by the DSP/BIOS II Configuration Tool. In addition to minimizing the target memory footprint, the DSP/BIOS II Configuration Tool provides the means for early detection of semantic errors through the validation of object attributes, prior to program execution. When the configuration tool is aware of all target program objects prior to execution, it can accurately compute and report such information as the total amount of data memory and stack storage required by the program.

2.1.1.1 Achieving Scalability

Since DSP/BIOS II is organized as a set of modules, and packaged as a run-time library, developers can select and link with their application only those modules they want to use. DSP/BIOS II is scalable, since the developer scales the amount of DSP/BIOS II to use and include.

2.1.2 DSP/BIOS II Kernel Object Viewing Support in CCStudio

Another element introduced within the DSP/BIOS II environment depicted earlier, is the integrated support within the CCStudio debugger (Figure 4). This support allows developers to view the current configuration, state, and status of DSP/BIOS II objects running in the target system. This hosted tool allows developers to view both static and dynamic DSP/BIOS II kernel objects in the target application, referenced by the user-defined name and location.

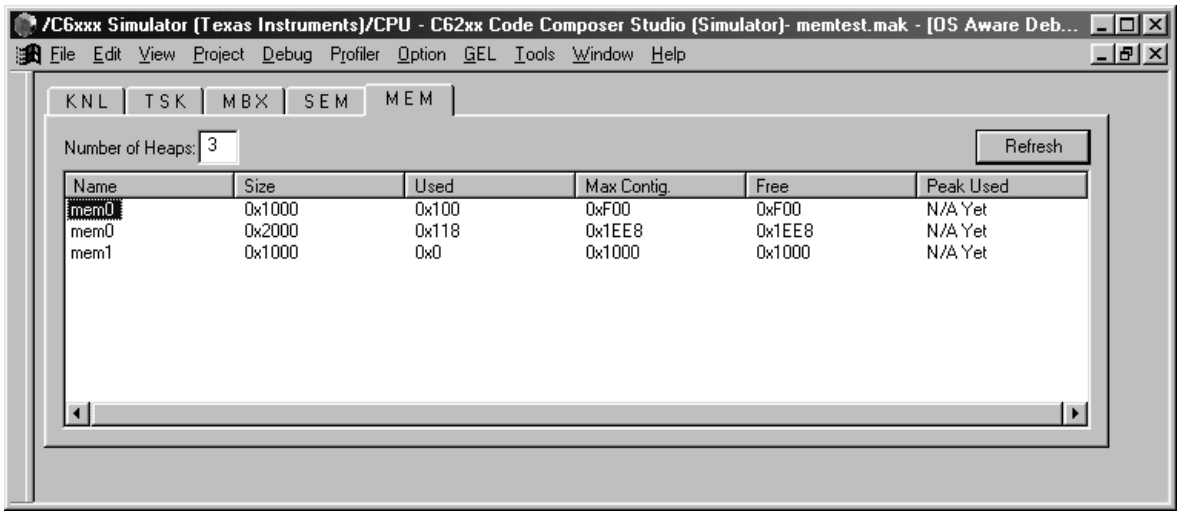

Figure 4. Code Composer Studio Debugger

The DSP/BIOS II kernel-aware extensions complement the familiar program debugger with a series of hosted utilities that work hand-in-hand with the target-resident DSP/BIOS II kernel to enable program analysis. These tools are especially useful in debugging the program logic and control flow. Developers who build their applications using the DSP/BIOS II run-time services, are able to view the multitasking threads and other kernel objects at the same level of abstraction in which they are programming.

2.2 DSP/BIOS II Kernel

The DSP/BIOS II kernel implements run-time services that the target application program invokes through DSP/BIOS II APIs (see Table 1).

Individual DSP/BIOS II modules in general will manage one or more instances of a related class of objects, sometimes referred to as kernel objects, and will rely upon global parameter values to control their overall behavior.

Developers can statically declare and configure many of these objects using the DSP/BIOS II Configuration Tool. Developers may also declare and configure many of these objects dynamically within their programs.

Table 1. DSP/BIOS II Modules by Function

			Object Creations		Language Support	
Feature	**Module**		**STATIC**	**DYNAMIC**	**C routines**	**ASM routines**
Real Time Analysis and Data Capture						
Event Logging	LOG	Message Log Manager	x		x	x
Statistics Accumulation	STS	Statistics Accumulator Manager	x		x	x
Trace Control	TRC	Trace Manager	x		x	x
File Streaming	HST	Host I/O Manager	x		x	x
Real Time Data Exchange	RTDX	Target to Host Communication Manager	x		x	x
Hardware Abstraction						
On–Chip Timer	CLK	System Clock Manager	x		x	x
Hardware Interrutps	HWI	Hardware Interrupt Manager	x			x
Static Memory Management	MEM *	Memory Segment Manager	x			
Dynamic Memory Management	MEM **	Memory Segment Manager		X	x	
Device – Independent I/O						
Data Pipes	PIP	Data Pipe Manager	x		x	x
Data Streams	SIO	Stream I/O Manager	x	X	x	
Execution Thread Management						
Software Interrupts	SWI	Software Interrupt Manager	x	X	x	x
Periodic Functions	PRD	Periodic Function Manager	x		x	x
Tasks	TSK	Multitasking Manager	x	X	x	
Idle Loop	IDL	Idle Function and Processing Loop Manager	x		x	x
Inter – Thread Communnication and Synchronization						
Semaphores	SEM	Semaphore Manager	x	X	x	
Resource Locks	LCK	Resource Lock Manager	x	X	x	
Mailboxes	MBX	Mailbox Manager	x	X	x	
Queues	QUE	Queue Manager	x	X	x	
Other Services						
Atomic Functions (optimized and non–premptive)	ATM	Atomic Functions written in Assembly Language	N/A	N/A	x	
Error handling and program termination	SYS	System Services Manager	N/A	N/A	x	

*** Using the *Configuration Tool*, memory segments are defined and named.**

**** Once named, this module provides allocation and freeing services.**

2.2.1 Structuring Applications with DSP/BIOS II Execution Threads

When applications are organized as independent paths of execution, developers can place structure and order into them (see Figure 5). DSP/BIOS II execution threads are independent paths of execution that execute an independent stream of DSP instructions. An execution thread is a single point of control that may contain an ISR, subroutine, or a function call. For example, a hardware interrupt is a thread, and it performs the ISR when triggered.

By organizing DSP applications around execution threads, developers can structure their applications, apply appropriate priorities to each thread, and ensure their applications meet critical real-time deadlines.

Multithreaded applications can run on single processor systems by allowing higher-priority threads to preempt lower-priority threads. DSP/BIOS II provides 30 levels of priority, divided over four distinct classes of execution threads (see Figure 5). DSP/BIOS II also provides services to support the synchronization of, and communication between, execution threads. Multirate processing maps well onto multithreaded systems.

Each thread class has different execution, preemption and suspension characteristics (see Figure 6). Since all threads are fully preemptive, developers can integrate existing algorithms in DSP/BIOS II-based applications easily without having to change the algorithm source code. This is especially important in accommodating algorithms that must execute at differing rates. In addition, preemptive threading provides a simple way to guarantee that hard real-time threads get the CPU when required. Adding a new thread will not disrupt the correctness of the system. An important point to know is that the number of threads in the system has no bearing on the length of time it takes between a hardware interrupt and the thread it makes ready.

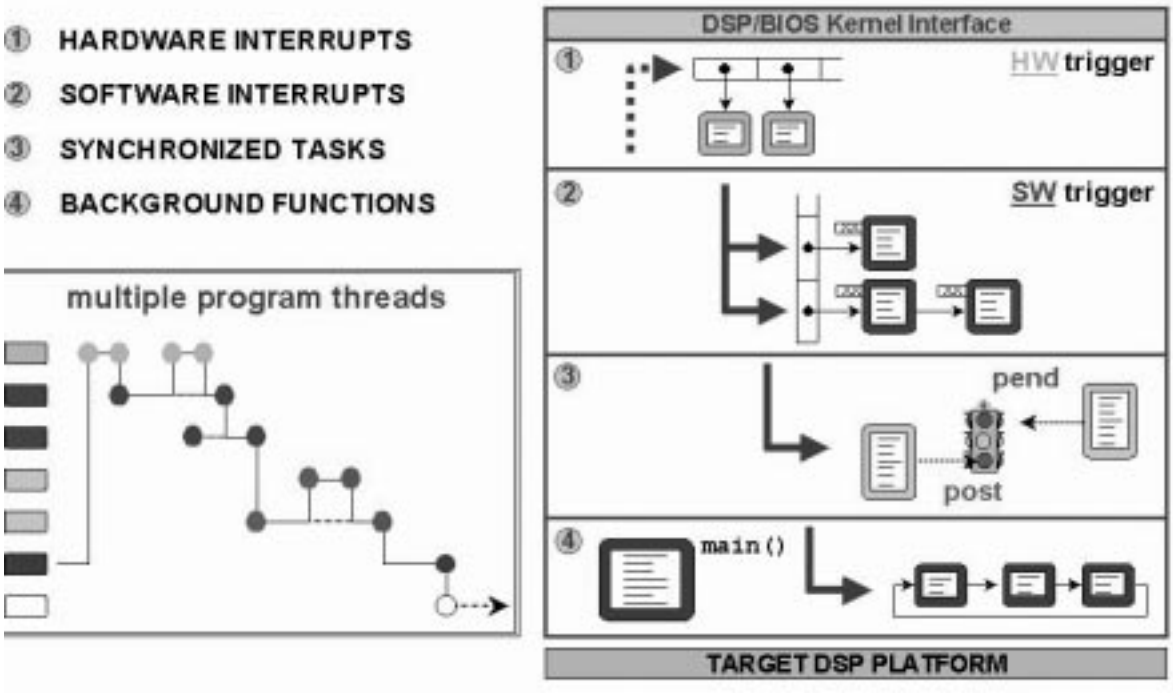

Figure 5. DSP/BIOS II Execution Threads

With the exception of the background idle processing thread, each thread type supports multiple levels of priority. DSP/BIOS II provides choices; it allows developers to use the optimum thread-types for their application and not bend their application to fit a certain model. Since there is no right solution for all applications, DSP/BIOS II developers have the flexibility to mix and match the objects in the run-time library that are best suited for the application. Moreover, since DSP/BIOS II is completely scalable, only those modules that have been selected link with the application, minimizing precious resource requirements.

2.2.1.1 Hardware Interrupts

In DSP/BIOS II, the HWI module manages a finite class of objects that correspond to individual hardware interrupts and are recognized by the underlying DSP platform. The HWI module provides run-time support to enable interrupt service routines that are associated with these objects to schedule execution of DSP/BIOS II software interrupts or synchronous tasks. The HWI module also provides run-time services to enable and disable executing hardware interrupts.

Through the DSP/BIOS II Configuration Tool, the HWI Manager is used to map the interrupt service routine to the hardware interrupt, locate the interrupt service tables anywhere in memory, and locate the HWI dispatcher. In DSP/BIOS II applications, developers are able to program ISRs in assembly language, or a mix of assembly and C.

A major benefit to using the HWI manager is device abstraction. The HWI module allows developers to use logical services to configure, manage and make use of hardware interrupts. When the application requires migrating to another device, the developer only needs to reassign the logical map to the new physical device.

2.2.1.2 Software Interrupts

As the name implies, this execution model is similar to the hardware ISR, except software interrupts are not associated to the physical DSP device; rather, they are instantiated in software. Like hardware ISR, software interrupts also execute in a run-to-completion mode, and they share the property of preemption. DSP/BIOS II software interrupts are priority-based, and support 14 levels of priority. The SWI module manages DSP/BIOS II software interrupts.

Software interrupts can only be preempted by a higher-priority software interrupt or a hardware interrupt. It is quite possible that an application will have multiple software interrupts sharing the same priority level. In this case, software interrupts of the same priority execute on first-come (posted), first-served basis. Figure 6 illustrates prioritized thread execution.

While external events trigger hardware interrupts, programs trigger software interrupts by calling SWI functions. Individual software interrupts are triggered for execution through the DSP/BIOS II API calls. These calls can be embedded in virtually any execution thread within the target.

On today's TMS320 devices, the DSP/BIOS II overhead is less than 1 microsecond for each interrupt routine that calls `SWI_post()`, suggesting minuscule impact within applications whose processing cycles fall under the 1 kHz threshold. Actual overhead is easily measured using the DSP/BIOS II RTA services and tools within CCStudio IDE.

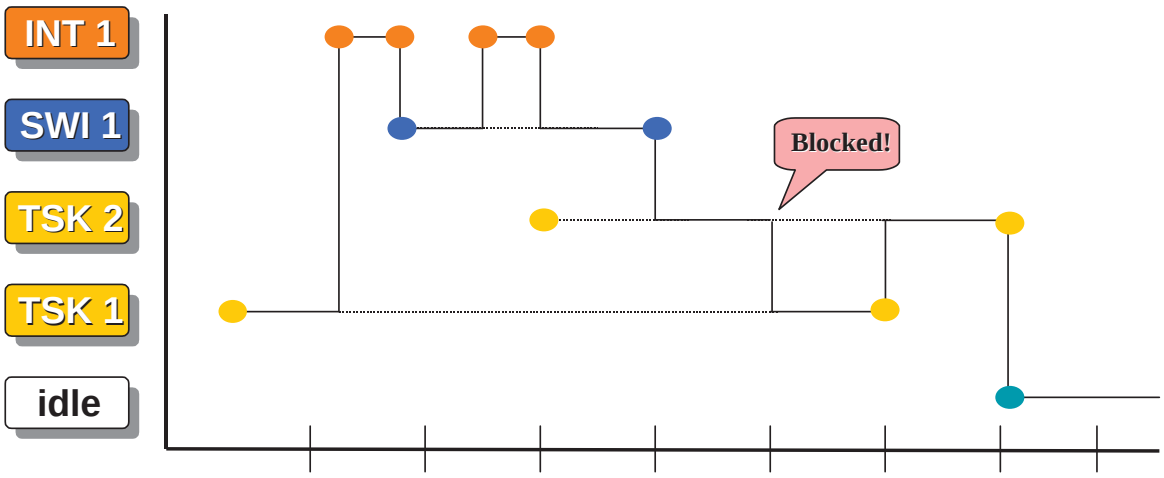

Figure 6. DSP/BIOS II Prioritized Thread Execution Model

With low overhead per context switch, DSP/BIOS II software interrupt objects also become an ideal mechanism for structuring programs that employ multiple algorithms executing at different rates (e.g., a telecommunication application where voice coding, tone detection, and echo cancellation typically process a common 8 kHz input stream using frames of differing duration that might range from 1-20 ms (milliseconds)). Through a technique called rate monotonic scheduling, binding algorithms with shorter deadlines to higher priority software interrupts ensures orderly interleaving of otherwise independent real-time threads that each contend for their respective allotment of processor cycles. Rate monotonic scheduling is possible since software-interrupt handlers will preempt one another on a strict priority basis.

2.2.1.3 Periodic Functions

DSP/BIOS II offers a special type of software interrupt, triggered by a periodic clock, that is used to schedule periodic functions or activities that must occur at periodic rates.

The PRD module in DSP/BIOS II manages these functions. DSP/BIOS II provides a system clock, a 32-bit counter that advances every time `PRD_tick`() is called. The timer interrupt can drive this system clock by calling `PRD_tick`(), however, other periodic events, such as data clocks, can also call `PRD_tick`(). The PRD manager allows developers to schedule the PRD software interrupt thread to execute functions at different rates. Developers can create an arbitrary number of PRD objects, each specifying a different period. However, since all PRD objects are driven from the same system clock, the rates are all integer multiples of the system clock or `PRD_tick`()s. PRD objects encapsulate a function, two arguments, and a period specifying the time between successive invocations. Since some applications need to have a function execute only once after a time delay, DSP/BIOS II periodic functions support a single-cycle or one-shot mode. PRD objects execute from a software interrupt, so they share the same stack as software interrupts.

2.2.1.4 Synchronized Tasks

Unlike the hardware and software interrupt model previously described, synchronized tasks in DSP/BIOS II (see Figure 7) are capable of suspension as well as preemption. Tasks will run to completion unless preempted or suspended. Synchronized tasks run at a lower priority than software interrupts, but above the idle background thread. DSP/BIOS tasks are priority-based, supporting 15 levels of priority, plus a suspended state. Tasks will be preempted by both hardware and software interrupts (SWI threads), and higher-priority tasks. In DSP/BIOS II, the TSK module manages and schedules synchronized tasks.

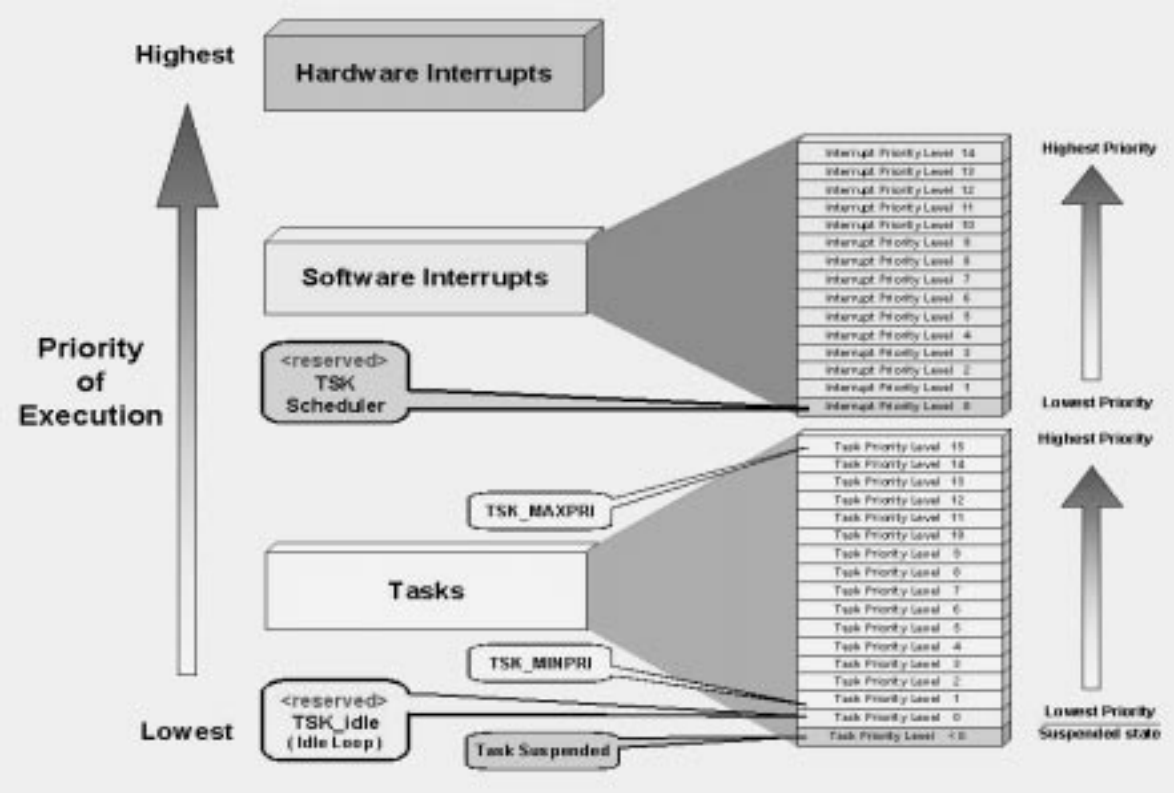

Figure 7. All DSP/BIOS II Execution Threads Prioritized

When a task is suspended, further thread execution suspends (blocks), waiting for a resource to become available, or an event to occur. Tasks can block themselves or any other task by:

- explicitly yielding (`TSK_yield()`),
- sleeping for some period of time (`TSK_sleep()`), or,
- waiting for a resource to become available or an event to occur that is synchronized by a semaphore (such as `SEM_pend()`, or `MBX_pend()`).

In contrast, both hardware and software interrupts cannot yield, sleep, or block — they always run to completion unless preempted.

DSP/BIOS II provides familiar kernel elements that form the basis of traditional concurrent processing. In traditional concurrent system designs, multithreaded applications structure around elements like tasks, semaphores, mailboxes, and message queues. Developers familiar with the programming models of VxWorks, PSOS, and Nucleus will find the components in DSP/BIOS II to be very similar.

2.2.1.5 Background Idle Processing

The Background Idle thread executes at the lowest priority in a DSP/BIOS II application. This thread runs continuously in the absence of any higher priority threads that need to run, such as interrupts or tasks.

Sometimes when organizing an application around independent threads of execution, one thread may be an operation that should only run when nothing else needs to run — in the background. This can be useful for polling non-real-time I/O devices or communications ports that are incapable of generating interrupts, monitoring system status or other operations that developers do not want to impact their real-time application. In fact, the communication between the RTA CCStudio plug-ins and the target application run in this background loop to ensure the host link does not interfere with the real-time application.

The IDL manager in the DSP/BIOS II Configuration Tool allows developers to insert functions to execute in the idle loop. The idle loop calls each function in the order listed, one at a time, and allows each function to run to completion. This process repeats cyclically.

Developers who wish to implement power-minimization operations can leverage the DSP/BIOS II kernel and background idle loop. The kernel allows the developer to simultaneously ensure system responsiveness to the highest-priority threads, and to idle the processor when there is nothing to do. The DSP/BIOS II kernel knows that it can idle the processor when it is in the idle loop. Developers can leverage this and inactivate the processor, or perform other power-minimization schemes.

2.2.2 DSP/BIOS II Real-Time Analysis (RTA)

The DSP/BIOS II Real-Time Analysis (RTA) features, shown in Figure 8, provide developers and integrators unique visibility into their application by allowing them to probe, trace, and monitor a DSP application during its course of execution. These utilities, in fact, piggyback upon the same physical JTAG connection already employed by the debugger, and utilize this connection as a low-speed (albeit real-time) communication link between the target and host.

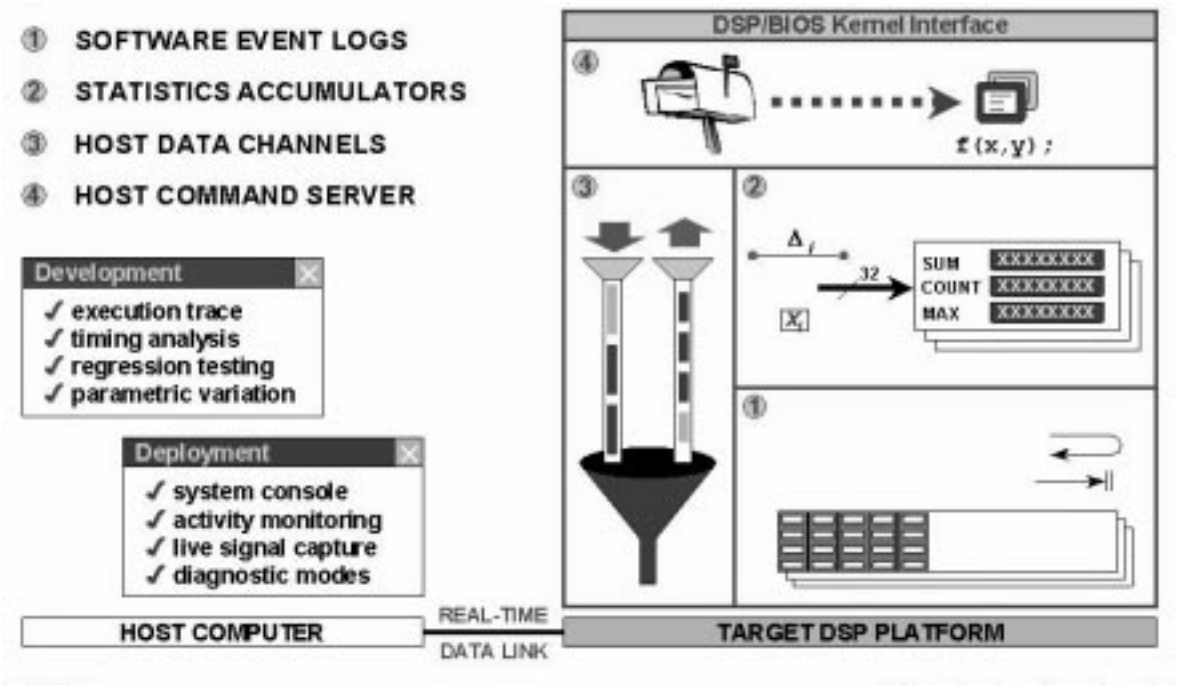

Figure 8. DSP/BIOS II Real-Time Capture and Analysis

DSP/BIOS II RTA requires the presence of the DSP/BIOS II kernel within the target system. In addition to providing run-time services to the application, DSP/BIOS II kernel provides support for real-time communication with the host through the physical link. By simply structuring an application around the DSP/BIOS II APIs and statically created objects that furnish basic multitasking and I/O support, developers automatically instrument the target for capturing and uploading the real-time information that drives the visual analysis tools inside CCStudio IDE. Supplementary APIs and objects allow explicit information capture under target program control as well. From the perspective of its hosted utilities, DSP/BIOS II affords several broad capabilities for real-time program analysis:

- **Message Event Logs** – capable of displaying time-ordered sequences of events written to kernel log objects by independent real-time threads, tracing the program's overall flow of control. The target program logs events explicitly through DSP/BIOS API calls or implicitly by the underlying kernel when threads become ready, dispatched, and terminated.
- **Statistics Accumulators** – capable of displaying summary statistics amassed in kernel accumulator objects, reflecting dynamic program elements ranging from simple counters and time-varying data values, to elapsed processing intervals of independent threads. The target program accumulates statistics explicitly through DSP/BIOS API calls or implicitly by the kernel when scheduling threads for execution or performing I/O operations.
- **Host Data Channels** – capable of binding kernel I/O objects to host files providing the target program with standard data streams for deterministic testing of algorithms. Other real-time target data streams managed with kernel I/O objects can be tapped and captured on-the-fly to host files for subsequent analysis.
- **Host Command Server** – capable of controlling the real-time trace and statistics accumulation in target programs. In effect, this allows developers to control the degree of visibility into the real-time program execution.

When used in tandem with the CCStudio standard debugger during software development, the DSP/BIOS II real-time analysis tools provide critical visibility into target program behavior at exactly those intervals where the debugger offers little or no insight — during program execution. Even after the debugger halts the program and assumes control of the target, information already captured through DSP/BIOS II can provide invaluable insights into the sequence of events that led up to the current point of execution.

Later in the software development cycle, regular debuggers become ineffective for attacking more subtle problems arising from time-dependent interaction of program components. The DSP/BIOS II real-time analysis tools subsume an expanded role as the software counterpart of the hardware logic analyzer.

This dimension of DSP/BIOS II becomes even more pronounced after software development concludes. The embedded DSP/BIOS II kernel and its companion host analysis tools combine to form the necessary foundation for a new generation of manufacturing test and field diagnostic tools. These tools will be capable of interacting with application programs in operative production systems through the existing JTAG infrastructure.

The overhead cost of using DSP/BIOS II is minimal, therefore instrumentation can be left in to enable field diagnostics, so that developers can capture and analyze the actual data that caused the failures.

2.2.3 Real-Time Data Exchange (RTDX)

Real-time data exchange (RTDX) allows system developers to transfer data between a host computer and DSP devices without interfering with the target application. This bi-directional communication path provides for data collection by the host as well as host interaction with the running DSP application. The data collected from the target may be analyzed and visualized on the host. Application parameters may be adjusted using host tools, without stopping the application. RTDX also enables host systems to provide data stimulation to the DSP application and algorithms.

RTDX consists of both target and host components. A small RTDX software library runs on the target DSP. The DSP application makes function calls to this library's API in order to pass data to or from it. This library makes use of a scan-based emulator to move data to or from the host platform via a JTAG interface. Data transfer to the host occurs in real-time while the DSP application is running.

On the host platform, an RTDX host library operates in conjunction with CCStudio. Data visualization and analysis tools communicate with RTDX through COM APIs to obtain the target data and/or to send data to the DSP application.

The host library supports two modes of receiving data from a target application: continuous and noncontinuous. In continuous mode, the data is simply buffered by the RTDX host library and is not written to a lot file. Continuous mode should be used when the developer wants to continuously obtain and display the data from a DSP application, and doesn't need to store the data in a log file. In noncontinuous mode, data is written to a log file on the host. This mode should be used when developers want to capture a finite amount of data and record it in a log file.

2.2.4 Hardware Abstraction

DSP/BIOS II provides APIs to access and configure certain hardware components independent of their physical implementation (abstraction). The hardware abstraction APIs simplify the configuration of these devices by providing a simple, logical user interface independent of the underlying device. By abstracting the device-dependent components like the on-chip timer (through CLK), and the hardware interrupts (HWI), migrating from one device or across ISAs is significantly simplified. Figure 9 illustrates the DSP/BIOS II hardware abstraction services.

DSP/BIOS II also supports memory management. Through the DSP/BIOS II Configuration Tool, developers define and name the physical memory segments, creating a logical memory map. DSP/BIOS II runtime APIs provide dynamic allocation and freeing of memory from within the application by using this logical memory map with the MEM module.

The device-independent I/O APIs provide services to perform frame-based or block-based data transfers, whether transferring data between the DSP and a peripheral or between multiple execution threads. Peripherals typically include I/O devices like CODECs or host systems. DSP/BIOS II supports both data pipes and data streams.

2.2.4.1 Real-Time Clock Services

The CLK module allows developers to configure the hardware-dependent components such as the on-chip timer, to provide the time-base for all application timer or periodic functions. The DSP/BIOS II Configuration Tool provides a simple, logical interface to set the on-chip timer to the desired time-base, typically 1 microsecond. Access to the physical timer registers is available if needed.

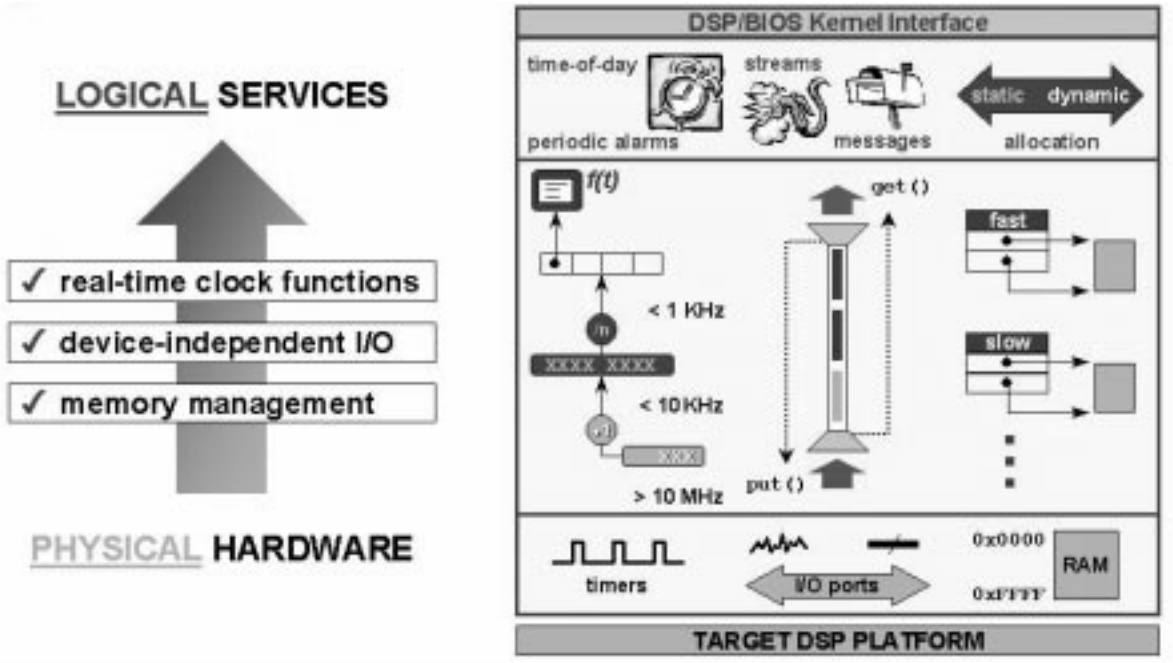

Figure 9. DSP/BIOS II Hardware Abstraction Services

The CLK module provides abstraction of the real-time clock with functions to access this clock at two resolutions. This clock can be used to measure the passage of time in conjunction with statistics accumulator objects, as well as to add timestamp messages to event logs. Both the low-resolution and high-resolution times are stored as 32-bit values. The low–resolution clock ticks at the timer interrupt rate and the clock's value is equal to the number of timer interrupts that have occurred. The high-resolution time is the number of times the timer counter register has been incremented and is a direct function of the DSP clock. High resolution time is useful for determining how long it takes the DSP to execute a series of instructions, using instruction cycles as the unit of measurement.

The Clock Manager allows developers to create an arbitrary number of clock functions. Clock functions are executed by the clock manager every time a timer interrupt occurs. These functions may invoke any DSP/BIOS II operations allowable from within hardware ISRs.

2.2.4.2 Memory Management

DSP/BIOS II supports memory management for TMS320 platforms. The DSP/BIOS II memory section manager allows developers to specify the memory segments required to locate the various code and data sections of a DSP/BIOS II application. The MEM module also provides a set of run-time functions used to allocate storage from one or more segments of memory. Developers use the DSP/BIOS II Configuration Tool to specify the memory segments, resulting in a logical memory map of the physical system. This allows developers to program at the logical level using the logical memory map specified through the Configuration Tool. Abstracting the physical memory map can simplify the process of of migrating the application to another hardware platform or next-generation TMS320 device.

Software modules in DSP/BIOS II that allocate storage at runtime use the runtime MEM functions. DSP/BIOS II modules use MEM to allocate storage in the segment selected for that module with the Configuration Tool.

2.2.4.3 Device-Independent I/O

Fundamental to all DSP applications is the need to acquire data, process it, and output the results. DSP applications typically process blocks of data simultaneously rather than a single datum. So these applications will move continuous blocks of data in from a source, process it, and output the results. Conceptually, this movement of data blocks forms a stream of data flowing in one direction from source to sink. These streams allow I/O and processing to occur at different rates due to the ability to manage multiple frames asynchronously. That is, while a device is currently filling one buffer with data, the DSP is processing a previously loaded buffer.

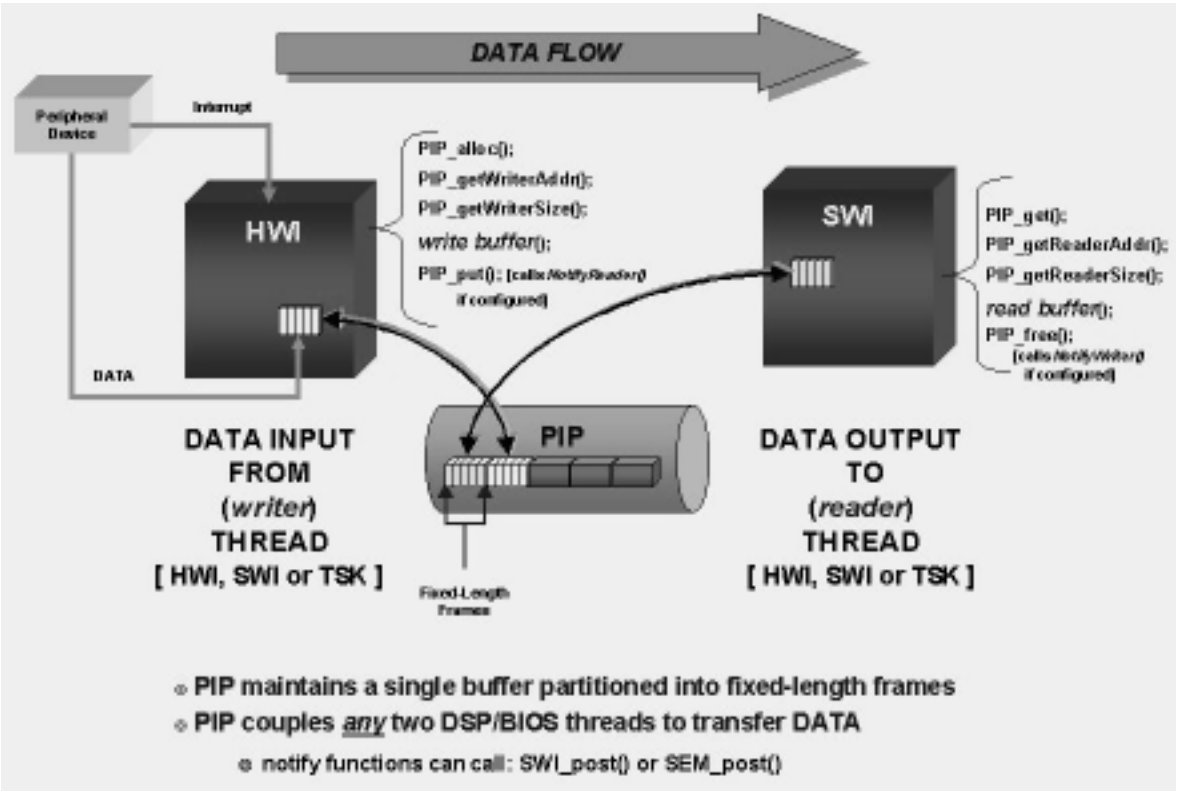

Figure 10. Data Pipes in DSP/BIOS II

DSP/BIOS II provides services to move blocks or frames of data by two primary mechanisms: data streams and pipes. DSP/BIOS II uses a buffer-passing mechanism that exchanges address pointers, transfers the buffers without any copying, and minimizes the amount of actual data being moved.

Data pipes are simple universal components that transfer frame-based data between a reader and a writer thread. Data pipes are small and efficient, and are statically bound at design time to optimize performance and minimize overhead.

Data streams also provide a device-independent, asynchronous, frame-based transfer mechanism for applications. Data streams may bind statically like data pipes, or they may bind dynamically during program execution. Data streams also offer flexibility in the buffering schemes to enable a broader range of application requirements. To achieve this flexibility, data streams rely on one or more underlying device drivers. The device driver encapsulates the device-dependent properties and methods. Device drivers can also perform other operations on data passing through them via a mechanism known as stacking device drivers. This class of device drivers offers the ability to pipeline processing operations such as data type conversions, scaling or filtering in the data path.

2.2.4.4 Data Pipes

Data pipes provide a code-efficient, universal data transfer mechanism ideally suited to communicate between the interrupt service routine and a deferred procedure such as a software interrupt or task. Data pipes are unidirectional; they pass buffers of data, or frames, between any two DSP/BIOS II execution threads. In DSP/BIOS II, data pipes are managed by the PIP module.

Data pipes support two modes of data transfer synchronization: polling and callback functions. Execution threads can poll the data pipe to synchronize data transfers; or for event-driven applications, the data pipe can call a notify-function (callback function) to alert that a buffer has become available. For example, a hardware ISR completes filling a buffer with incoming data and calls `PIP_put()` to give the buffer back to the data pipe. The data pipe manager will notify the thread attached at the other end of the pipe that there is a buffer available by calling the notify-function assigned to that end of the data pipe.

Data pipes maintain their own private memory pool, partitioned into a fixed-number of fixed-length buffers or frames. The data pipe *exchanges* these frames of data between a reader and the data pipe; and between the data pipe and the writer. If the application needs to transfer the buffer to another data pipe, the application will need to copy the data to the other data pipe. Data pipes must be declared and bound statically during design time using the DSP/BIOS II Configuration Tool.

2.2.4.5 Data Streams

Data streams support static declaration and binding; they also support run-time creation and binding. Data streams offer applications greater flexibility and structure than data pipes. Manipulations of the data streams include operations to open and close the streaming device, start, stop, and flush the stream, and to provide control. Data streams offer flexibility in both the buffering schemes and memory management. Buffers may or may not be private to the data stream.

In DSP/BIOS II, the SIO module manages data streams at the application level. To the application, all devices appear the same due to the device–independent abstraction inherent in SIO. Complementing and interacting with SIO is the DEV module that manages the device–dependent drivers. Figure 11 shows the transfer of data between the application and the device through device-dependent and device-independent drivers.

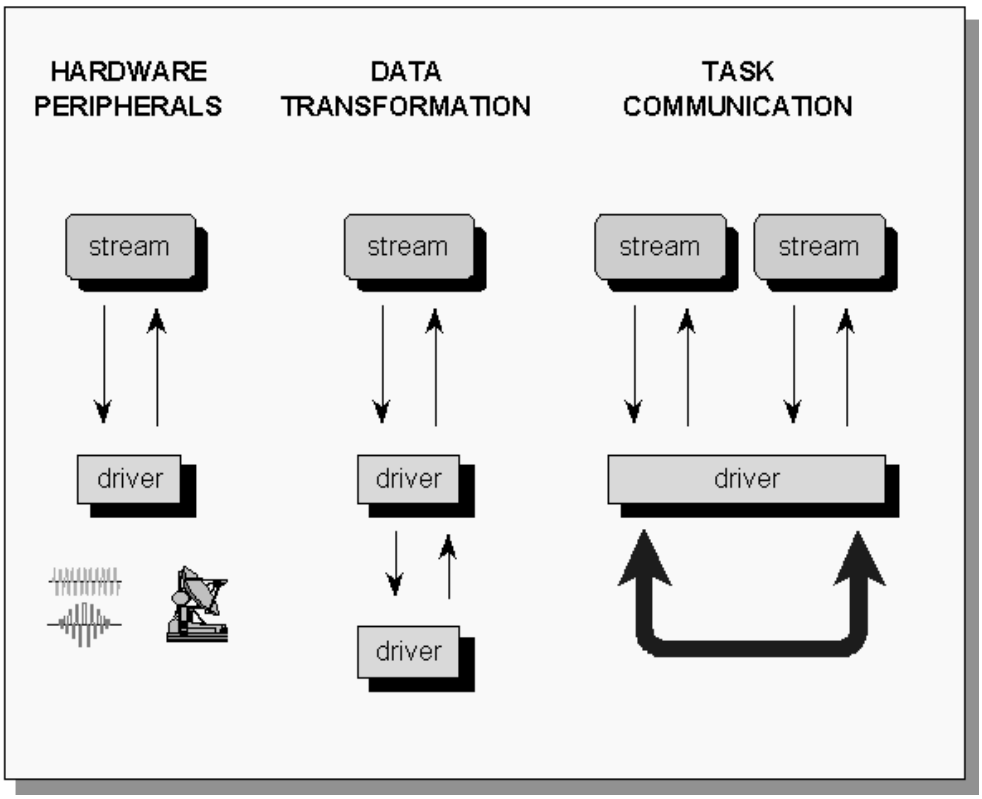

Figure 11. Data Stream in DSP/BIOS II

2.2.4.6 SIO Device drivers

The underlying device driver can perform a variety of functions. DSP/BIOS II supports two classes of device drivers: terminating and stacking.

Terminating drivers perform the classical I/O with peripherals such as CODECS. These device drivers often contain interrupt service routines (ISRs) to interact with the peripheral, however I/O by other means such as polling is equally permissible.

Stacking drivers are a special class of drivers that perform in-line, pipelined processing. A typical use of stacking device drivers is to implement operations such as scaling or filtering. One class of stacking device driver is a copying driver. These drivers can support variations in data size, buffer size, and buffer lengths. These are useful for performing data format conversions such as fixed-point to floating-point, or 14-bit audio to 16-bit data. If encapsulation and reuse of these operations is beneficial, they become candidates for implementation as a stacking device driver. Figure 12 illustrates how stacking device drivers are used to provide pipeline processing.

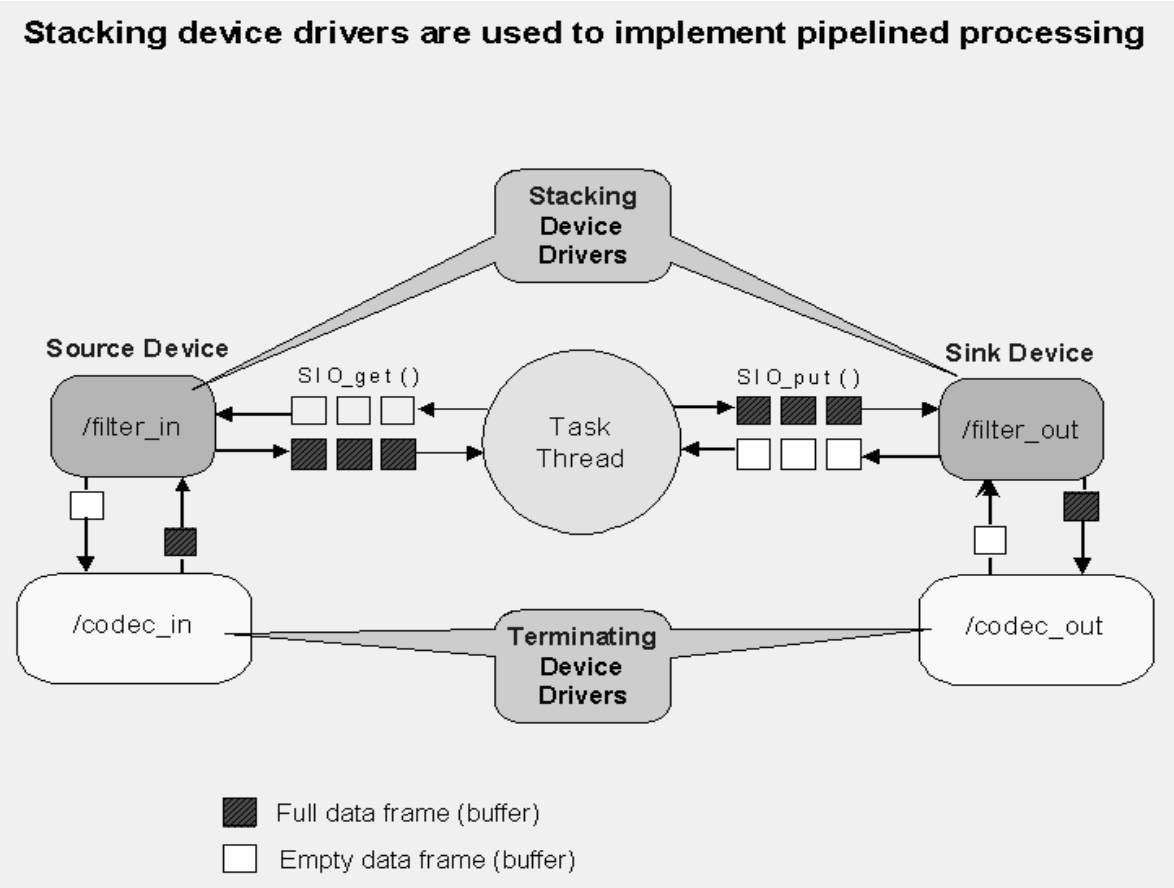

Figure 12. Stacking Device Drivers

3 Sample Application

The DSP/BIOS II Audio example, packaged with CCStudio, is shown in Figure 13. The Audio example program contains four DSP/BIOS II execution threads. The first two consist of the hardware interrupt for the serial port, and the audio I/O software interrupt thread, whose execution periodicity depends on the buffer rates. The other two consist of the periodic function software interrupt that will execute the periodic function and a hardware interrupt that services the DSP/BIOS II real-time clock.

The audio I/O threads collect audio data from a CODEC, copies it to an output buffer, and sends it back out to the CODEC. This example is not unlike many traditional embedded applications that must collect input data, do some processing on it, then output the processed data. For simplicity, the Audio example simply copies input buffers to output buffers. In this example, the CODEC samples the audio input at16kHz (the hardware interrupt rate); the filtering operations must occur every 4 ms; thus, the data pipe is configured to hold buffers that contain 4 ms of audio data, or 64 data points at the 16kHz rate.

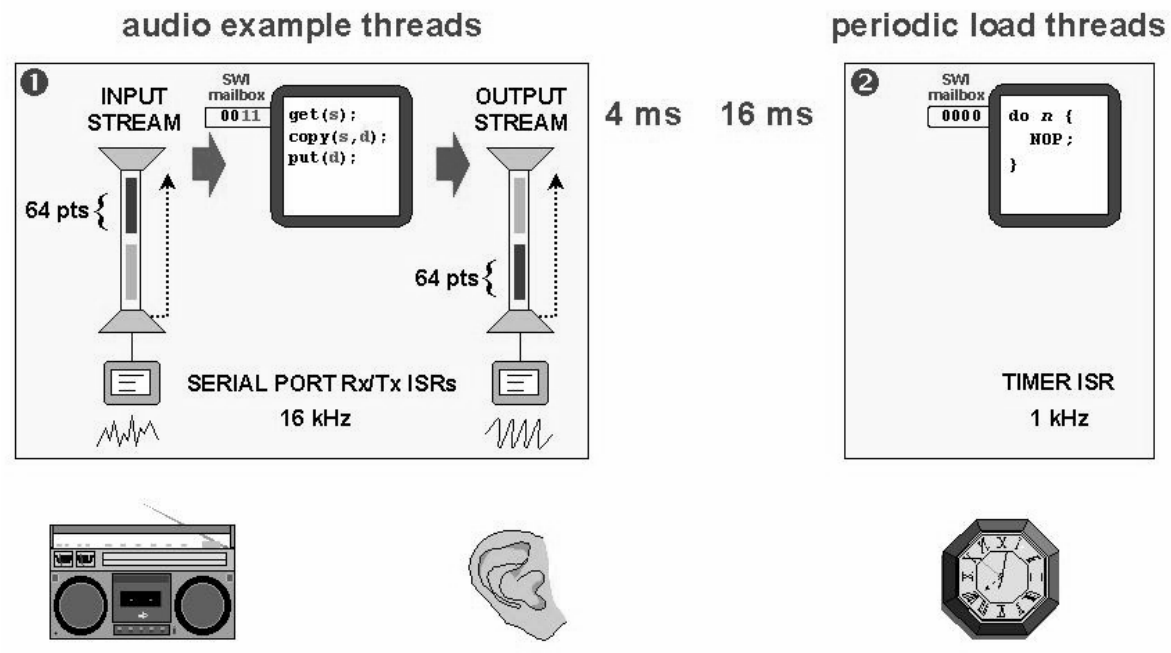

Figure 13. Audio Filter

The copy operation at the heart of this particular example executes once every 4 ms within the context of a single DSP/BIOS II SWI object. The software interrupt is triggered when the next full input, *and* empty output frames are ready for processing. This particular implementation relies upon statically configured data pipe callbacks to the kernel function SWI_andn(). This call clears individual bits in the software interrupt's mailbox representing this pair of triggering conditions (see the SWI mailbox in Figure 13) . Once dispatched, the SWI handler resets the mailbox to its initial non-zero value before retrieving descriptors for the next set of frames, and then invoking the algorithm itself.

```
AudioFilter( inputPipe, outputPipe )

{       PIP_get(inputPipe);         /* dequeue full frame */
        PIP_alloc(outputPipe);      /* dequeue empty frame */

        copy algorithm;             /* read/write data frames */
        PIP_free(inputPipe);        /* recycle input frame */

        PIP_put(outputPipe);        /* enqueue output frame */

        return;                     /* wait for next frame pair */

}
```

At the opposite ends of the input and output pipe from the audio filter lie a pair of interrupt service routines. These routines manage the underlying hardware peripherals that ultimately produce and consume the data streams processed by the algorithm. In spite of the differences in how these routines are implemented, these interrupt threads invariably exchange full and

empty data frames with the SWI thread through analogous pairings of PIP operations. Since PIP_free() and PIP_put() will implicitly invoke SWI_andn(), which in turn will post the audio filter software interrupt when its mailbox converges to 0, this segment of the interrupt routine must be appropriately bracketed with HWI_enter and HWI_exit macros to ensure the DSP/BIOS II kernel gains control upon return and performs the necessary context switch.

```
inputInterrupt()
 {
       service hardware;
       if (current frame is full) {
             HWI_enter;
             PIP_put(inputPipe);        /* enqueue current full frame */
             PIP_alloc(inputPipe);      /* dequeue next empty frame */
             HWI_exit;                  /* dispatch pending signals */
       }
}
outputInterrupt()
 {
       service hardware;
       if (current frame is empty) {
             HWI_enter;
             PIP_free(outputPipe);      /* enqueue current empty frame */
             PIP_get(outputPipe);       /* dequeue next full frame */
             HWI_exit;                  /* dispatch pending signals */
       }
}
```

The second part of the audio example is an operation that executes periodically based upon a real-time clock. This is not unlike many traditional embedded applications that must do some processing at periodic rates. For simplicity in our example, the purpose of this operation is to simulate a load on the DSP by executing NOP instructions. Introducing this load periodically simulates the interaction of multiple threads executing on the system. The point of the example is to demonstrate the effects of thread priorities on real–time systems. If the load function performs enough NOPs and is assigned a higher priority than the Audio function, then the audio signal will break–up when the load function preempts the audio function from meeting its real-time deadlines. This will even occur if the total DSP load is far less than 100%.

Periodic functions execute at the rates that are integer multiples of the PRD_tick(), which is typically synchronized to the data, but for our example, it is based upon the real-time clock. The developer through the DSP/BIOS II Configuration Tool specifies the PRD tick binding and the periodic function execution rates. When the specified number of PRD_tick()s occurs, DSP/BIOS II triggers the PRD thread. The PRD thread is a special software interrupt. In our example, the PRD_tick() rate is 1 ms, and the PRD thread has one function specified to trigger every 16 PRD_tick()s or 16 ms.

4 System Performance

We can estimate the amount of DSP/BIOS II overhead in terms of CPU load in any application. This is possible since all DSP/BIOS II operations are visible to the developer. That is, the developer specifies which DSP/BIOS II components and function calls to include into the application, either in the Configuration Tool, or explicitly in the code. The developer needs only to compute the sum of the components and frequency of occurrence to determine the overhead analytically. By using the RTA tools in CCStudio, developers may also directly measure the overhead on their specific hardware platform.

To estimate the overhead in DSP/BIOS II applications, the developer must first identify all the DSP/BIOS II components and API calls within the application. In our sample application audio I/O example, the DSP/BIOS II components are:

- one HWI object mapped to the Audio CODEC,
- one SWI object to do the processing (copy) operation and,
- two Data Pipes; one for input, one for output.

The component overhead in instruction cycles may be taken from the DSP/BIOS II performance data. To process a single buffer of audio data requires the total overhead of 1351 cycles on a C6000 as listed in Table 2. The processing period is 4 ms, so the frequency of occurrence is 250 times per second. Therefore, the total number of cycles in one second, attributed to DSP/BIOS II overhead running the audio thread on a C6000 DSP is 337,750 or 0.33775 MIPS. On a 200 MHz C6000 DSP, this equates to 0.17% CPU load.

To calculate the amount of memory consumed by DSP/BIOS II, the developer again needs to identify the DSP/BIOS II components and API calls in the program. By summing the components, the developer can estimate the memory usage, both data and program. By using the memory map from the application, the exact amount can be determined.

In a similar fashion, developers can analytically determine the overhead attributed to DSP/BIOS II. However, since it is the nature of software to change over time, analytical calculation can be tedious. The real-time analysis tool provided by DSP/BIOS II allow developers to measure the overhead directly. Finally, since developers can chose the amount of DSP/BIOS II to use and include in their applications, they have full control over the overhead.

Table 2. DSP/BIOS II Overhead in C6000 Cycles

Input Pipe		**Output Pipe**		**Hardware Interrupts**	
PIP_alloc	94	PIP_get	94	HWI_enter	
PIP_put	91	PIIP_free	89	HWI_exit	284
	185		183	1 time (buffer full)	284
Audio Function		**Data Pipes**			
PIP_get	94	SWI_andn	114		
PIIP_free	89	SWI_andn	217		
PIP_alloc	94		331		
PIP_put	91				
	368			Total:	1351

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.