Using the DSP/BIOS Kernel in Real-Time DSP Applications

David Dart (updated by Shawn Dirksen) *Software Development Systems*

ABSTRACT

The DSP/BIOS™ kernel provides system software capabilities and services to enable the development of more advanced and complex applications, and provides support for traditional concurrent-system design paradigms.

The DSP/BIOS kernel provides real-time system services and kernel components that provide preemptive multitasking services common in many embedded microkernels. These services enable applications to extend the event-driven, interrupt service paradigms of DSP/BIOS to take advantage of traditional multitasking services, mailboxes, queues, semaphores, and resource protection locks. The DSP/BIOS kernel integrates these traditional concurrent-system kernel services into a fully scalable kernel.

The DSP/BIOS kernel supports the data pipes and an alternative I/O model known as data streams, giving developers the opportunity to select the optimum I/O mechanism for their application.

The DSP/BIOS kernel allows static configuration of the kernel object programming model as well as dynamic kernel object creations. The DSP/BIOS kernel allows runtime memory management providing dynamic memory allocations and de-allocations. Applications can now manage and configure resources dynamically, enabling developers to build self-configuring and more complex applications whose mix of functions changes over its lifetime.

Contents

List of Figures

List of Tables

1 DSP/BIOS Overview

When Texas Instruments introduced the DSP/BIOS kernel with Code Composer Studio IDE (CCStudio), developers secured the opportunity to leverage decades of real-time DSP software experience. By building applications on top of the powerful, yet diminutive system services afforded by DSP/BIOS, developers focus their efforts on their applications rather than the underlying infrastructure. Furthermore, with the underlying infrastructure instrumented for real-time trace and analysis, developers use these services to develop applications that are more reliable.

The DSP/BIOS kernel is an evolutionary next step in the DSP software infrastructure provided by Texas Instruments. The features of the DSP/BIOS kernel enable both traditional and new programming methods and paradigms. With the introduction of DSP/BIOS, the kernel and I/O services extend to encompass and enable a greater variety of application demands.

The DSP/BIOS kernel provides a host-side utility to assist developers during application debug. The DSP/BIOS Kernel Object Viewing CCStudio plug-in provides developers visibility into kernel objects during program execution. Developers can use this tool to view the current state of the executions threads or the properties of kernel objects used within the application.

Some DSP applications may require processing commensurate with the data presented. An application may need to create and configure a processing thread on-the-fly to accommodate the incoming data. For example, voice applications in Internet Telephony often need to negotiate and dynamically attach the correct Vocoder to the incoming voice channel. Applications of this class take on a dynamic nature. Both the instantiation and configuration of the data path are dynamic. To accommodate dynamic system requirements, the DSP/BIOS kernel supports dynamic real-time kernel object creation and configuration. With DSP/BIOS, developers choose when to create execution threads and data I/O channels. For optimal performance, applications might create most kernel objects statically up front with the configuration tool, and create others dynamically on-the-fly, commensurate with the application's run-time demands.

This document describes the DSP/BIOS kernel and its services. The first section of this document presents the feature-function matrix to provide an overview of the modules in the DSP/BIOS kernel. The rest of the paper discusses these modules in more detail.

2 DSP/BIOS Kernel API Overview

The DSP/BIOS kernel provides real-time system services and kernel components for TMS320C5000™ and TMS320C6000™ DSP applications. These services and components are available through a combination of design-time declarations using the DSP/BIOS Configuration Tool, and a run-time library.

Some DSP/BIOS components require declaration and configuration within the configuration tool and have no Application Programming Interfaces (APIs) exposed. Other services expose a set of APIs to allow developers to access them within their application. The DSP/BIOS kernel implements run-time services that the target application program invokes through the DSP/BIOS APIs.

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

The DSP/BIOS modules are grouped into six functional classifications (see Table 1):

- System Services
- Instrumentation
- Scheduling
- Synchronization
- Input/Output
- Chip Support Library (CSL)

DSP/BIOS configuration tool provides a visual editor to define global properties, the system memory map, interrupt vector table, and program the on-chip timer. The DSP/BIOS kernel offers APIs that provide dynamic allocation and freeing of memory during run time exposed in the **MEM** module.

The DSP/BIOS kernel provides basic system services for program error handling and termination patterned after some standard C-library services grouped under **Miscellaneous System Services** modules. The **SYS** module manages these functions. SYS also includes variations on `printf()`.

Atomic functions are non-preemptive operations that execute with interrupts disabled. Operations include AND, OR, INC, DEC, SET and CLEAR. The **ATM** module manages these functions.

The **Instrumentation or Real-Time Analysis** (RTA) modules enable developers to take advantage of real-time interaction and diagnostics with the DSP application while it is executing in real time. These modules provide APIs to programmatically instrument user-developed components for real-time interaction and diagnostics. The **LOG**, **STS**, and **TRC** modules manage these functions.

The DSP/BIOS scheduling modules provide developers with four distinct classes of execution threads. Each thread class provides different execution, preemption and suspension characteristics. The DSP/BIOS kernel supports the two high-priority interrupt threads and the background idle processing loop. The **HWI**, **SWI**, and **IDL** modules manage these kernel execution threads. In addition, the DSP/BIOS kernel provides a multitasking thread class, Synchronized Tasks, capable of synchronously suspending and resuming their execution at any point in their execution. Tasks are also capable of arbitrarily adjusting their own (or any other task's) priority during program execution. Synchronized tasks form the basis of traditional concurrent system design. The **TSK** module manages this kernel execution thread.

TMS320C5000 and TMS320C6000 are trademarks of Texas Instruments.

The DSP/BIOS kernel provides logical interfaces to access and configure certain hardware components independent of the hardware implementation. The **Hardware Abstraction** modules simplify migrating applications across supported ISAs by abstracting the device-dependent components such as the system memory map, on-chip timer, and hardware interrupts.

The **Synchronization or Inter-Thread Communication** modules support the multitasking services provided in the DSP/BIOS kernel. Primary among these are semaphores. Tasks use semaphores to synchronize resource access. Typical examples of resource synchronization include exchanging buffers of data with peripherals, or arbitrating access to shared memory. Semaphores also control synchronizing the execution of multiple tasks. The **SEM** module manages semaphores. The **LCK** module provides shared-resource arbitration or mutual exclusion.

Data queues provide inter-thread communication or messaging, without synchronization. Queues are available for all DSP/BIOS threads. Mailboxes are similar to data queues, except they provide synchronization, and are ideally suited for inter-task communication. The **QUE** module manages data queues, and the **MBX** manager manages the mailbox module.

The **Device-Independent Input/Output** modules provide services to perform data transfers, whether transferring data between the DSP and a peripheral or between multiple execution threads. The DSP/BIOS kernel supports the data pipes and an alternative I/O model known as data streams. Data pipes are small and fast universal components that transfer data between a reader and a writer thread. Data streams add greater flexibility in the buffering schemes to enable broader application requirements. Data streams rely on one or more underlying device drivers. The device driver model encapsulates the *device-dependent* properties and methods. Device drivers are also capable of performing processing operations on data passing through them by a mechanism known as stacking. Stacking device drivers offer the ability to pipeline processing operations such as data type conversions, scaling or filtering in the data path. The **PIP** and **SIO** modules manage data transfer services in target applications. **SIO** also has an accompanying device driver module, **DEV,** which sources or sinks data with **SIO**.

Critical to the real-time analysis is the ability to transfer data between the host and the target. The DSP/BIOS kernel also provides APIs to manage the transfers. The **HST** and **RTDX** modules manage these functions.

The **Chip Support Library (CSL)** provides a C-language interface for configuring and controlling on-chip peripherals. It consists of discrete modules that are built and archived into a library file. The CSL module is the top-level API module. The primary purpose of the CSL module is to initialize the library. The CSL_init( ) function must be called once at the beginning of your program before calling any of the other CSL API functions.

The configuration tool is used to program the peripherals such as the Direct Memory Access (DMA), Multichannel Buffered Serial Port (McBSP), and other peripherals as well.

Refer to Table 1 for a full listing of the DSP/BIOS Modules. 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 *TMS320 DSP/BIOS User's Guide* (SPRU423).

Table 1. DSP/BIOS Features and Function Matrix

Description	Module	Object Creations		Language Support	
		Static	Dynamic	C Routines	ASM Routines
System Services					
Global Settings		X			
Static Memory Segment Manager	MEM†	X			
Dynamic Memory Segment Manager	MEM‡		**X**	X	
System Services Manager	SYS	N/A	**N/A**	X	
Atomic Functions (optimized and non-preemptive)	ATM	N/A	**N/A**	X	
Instrumentation					
Message Log Manager	LOG	X		X	X
Statistics Accumulator Manager	STS	X		X	X
Trace Manager	TRC	X		X	X
Scheduling					
System Clock Manager	CLK	X		X	X
Periodic Function Manager	PRD	X		X	X
Hardware Interrupt Manager	HWI	X			X
Software Interrupt Manager	SWI	X	**X**	X	X
Multitasking Manager	TSK	X	**X**	X	
Idle Function and Processing Loop Manager	IDL	X		X	
Synchronization					
Semaphore Manager	SEM	X	**X**	X	
Resource Lock Manager	LCK	X	**X**	X	
Mailbox Manager	MBX	X	**X**	X	
Queue Manager	QUE	X	**X**	X	
Input/Output					
Target-to-Host Communication Manager	RTDX	X		X	X
Host I/O Manager	HST	X		X	X
Data Pipe Manager	PIP	X		X	X
Stream I/O and Device Driver Manager	SIO/DEV	X	**X**	X	
Chip Support Library (CSL)					
Direct Memory Access	DMA	X	**X**	X	
Enhanced Direct Memory Access (TMS320C6x™ only)	EDMA	X	**X**	X	
External Memory Interface (TMS320C6x and TMS320C55x™ only)	EMIF	X	**X**	X	
Multichannel Buffered Serial Port	McBSP	X	**X**	X	
Timer Device	TIMER	X	**X**	X	
Expansion Bus (TMS320C6x only)	XBUS	X	**X**	X	
Instruction Cache (TMS320C55x only)	CACHE	X	**X**	X	
General Purpose Input/Output (TMS320C5x™ only)	GPIO	X	**X**	X	
Clock Generator (TMS320C5x only)	PLL	X	**X**	X	
Watchdog Timer Device (TMS320C54x™ only)	WDTIMER	X	**X**	X	

† Using the Configuration Tool, memory segments are defined and named.
‡ Once named, this module provides allocation and freeing services.

TMS320C6x, TMS320C55x, TMS320C5x, and TMS320C54x are trademarks of Texas Instruments.

3 DSP/BIOS Kernel

Building upon the real-time foundation of the DSP/BIOS kernel provides capabilities and support for traditional concurrent-system design paradigms. This support is a combination of software modules, allowing both static and dynamic instantiation, and configuration of kernel objects.

As mentioned, the DSP/BIOS kernel provides multitasking thread class — synchronized tasks. Tasks are independent threads of execution running at lower priority than software interrupts, but above the idle loop (see Figure 1). The DSP/BIOS kernel introduces semaphores to synchronize the scheduling of tasks and access to resources. The DSP/BIOS kernel makes use of semaphores internally within a whole new class of kernel objects including mailboxes, resource locks, and data streams.

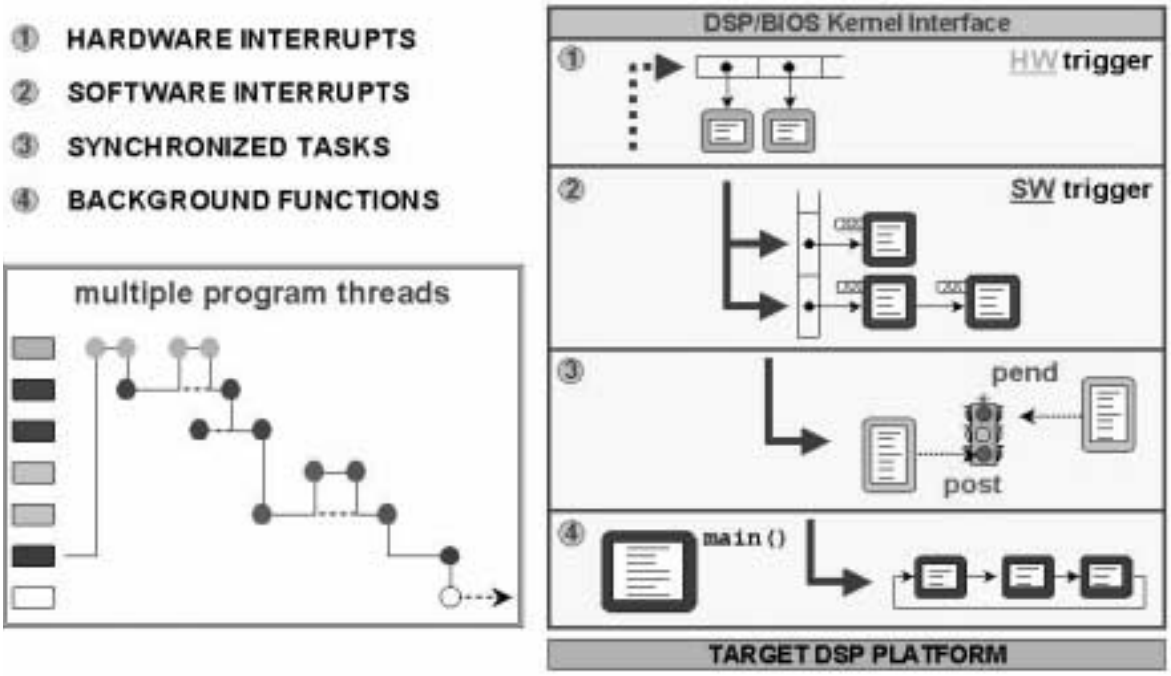

Figure 1. DSP/BIOS Kernel Execution Threads

The DSP/BIOS kernel provides a data transfer module using frame-based data streams with semaphore synchronization and programmable buffer management. Integral to this data streaming module is a corresponding device driver module. This device driver module also supports pipelined operations using a mechanism known as *stacking*.

The DSP/BIOS kernel also supports dynamic creation and configuration of many kernel objects. To enable this, the DSP/BIOS kernel provides support for dynamic memory allocation and deallocation. In the DSP/BIOS kernel, not only the configuration tool declares and configures all the kernel objects, but it also allows the creation and configuration of many of the kernel objects during program execution. This enables a frontier of applications that need flexible and dynamic services. Applications can dynamically create, configure, or reconfigure the processing threads based on the data presented. For example, telephony applications can create and configure voice channels dynamically with each incoming call, then release them when the call terminates.

3.1 Execution Threads in the DSP/BIOS Kernel

Real-time systems deployed on digital signal processors (DSPs) typically require the use of hardware interrupts to trigger processing of external events. Once triggered, the DSPs will invoke an interrupt service routine to perform the required operations needed to service the external event, such as acquiring or delivering data to a peripheral device. These interrupt service routines, or ISRs, will preempt the normal processing operations of the DSP, and once activated, the ISR will execute until it is either finished, or it is preempted by another hardware interrupt if so enabled. The ISR executes in a model known as *run-to-completion*. This execution model does not provide a mechanism to stop execution once started, except by preemption. In the DSP/BIOS kernel, the HWI module manages the hardware interrupts. The HWI module provides mechanisms to enable and disable executing hardware interrupts, locate the interrupt service tables, and locate the HWI dispatcher. HWI executes in the context of the application, and uses the application's stack.

Similar to the hardware interrupt is an execution model named software interrupts (SWI). As the name implies, this execution model is similar to the hardware interrupt, except all software interrupts execute at a lower priority than hardware interrupts, and a software interrupt cannot interrupt itself. DSP/BIOS software interrupts are priority-based, providing 14 levels of priority (see Figure 2).

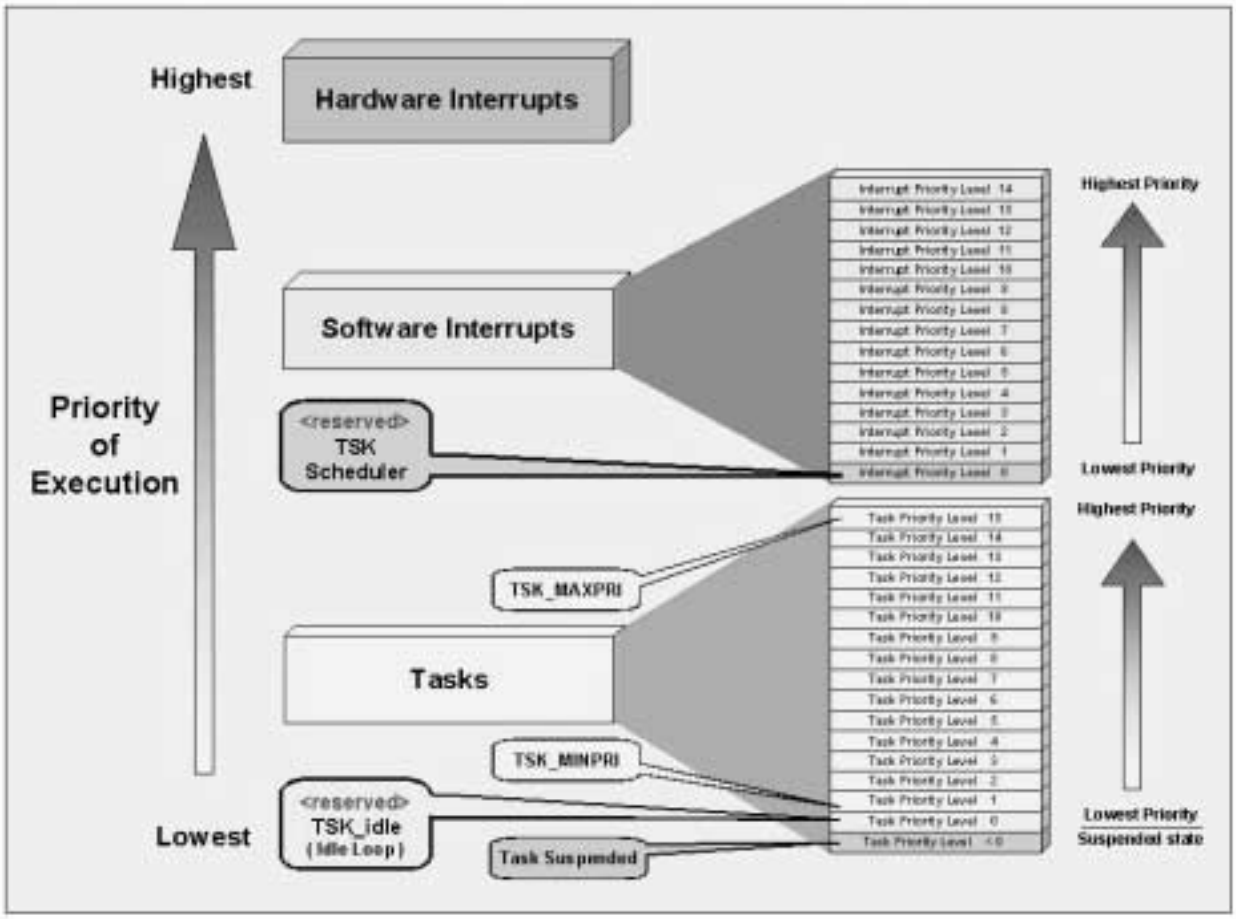

Figure 2. DSP/BIOS Thread Priorities

Like the hardware ISR, the software ISR also executes in a run-to-completion mode. 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 a first-come (posted), first-served basis.

In the DSP/BIOS kernel, the SWI module manages software interrupts. The SWI manager automatically saves processor registers on a stack before executing the software interrupt service routine. SWI executes in the context of the application, and uses the application's stack. Software interrupts execute using the same stack as hardware interrupts so the developer does not need to allocate an additional amount of memory to contain a separate SWI stack.

3.2 DSP/BIOS Execution Thread Model With Synchronized Tasks

The DSP/BIOS kernel provides synchronized task threads (**TSK**). Unlike the hardware and software interrupt model previously described, synchronized tasks in the DSP/BIOS kernel run either until they are completed, preempted, or until they are blocked awaiting a resource or some event to occur. Tasks suspend or terminate their execution themselves by explicitly yielding, sleeping or pending on a resource or event synchronized by a semaphore. In contrast, both hardware and software interrupts cannot yield, sleep, or suspend (block) — they always run to completion unless preempted.

DSP/BIOS tasks form the basis of traditional concurrent processing by partitioning applications into independent threads of execution, synchronized with semaphores or the system clock. Tasks dynamically modify their execution through semaphore synchronization services.

In the DSP/BIOS kernel, the TSK module manages and schedules tasks. DSP/BIOS tasks are priority-based, supporting 15 levels of priority plus a suspended state (negative task priority level). The lowest priority level (0) executes the Idle-loop (see Figure 2). The DSP/BIOS kernel permits both static and dynamic creation of tasks. However, only tasks created statically using the Configuration Tool are visible in the real-time analysis execution graph.

Each TSK task object (shown in Figure 3) is always in one of four possible states of execution:

1. Running, which means the task is the one actually executing on the system's processor;
2. Ready, which means the task is scheduled for execution subject to processor availability;
3. Blocked, or suspended, which means the task cannot execute until a particular event occurs within the system; or
4. Terminated, which means the task is "terminated" and does not execute again.

Tasks are scheduled for execution according to a priority level assigned to the application.

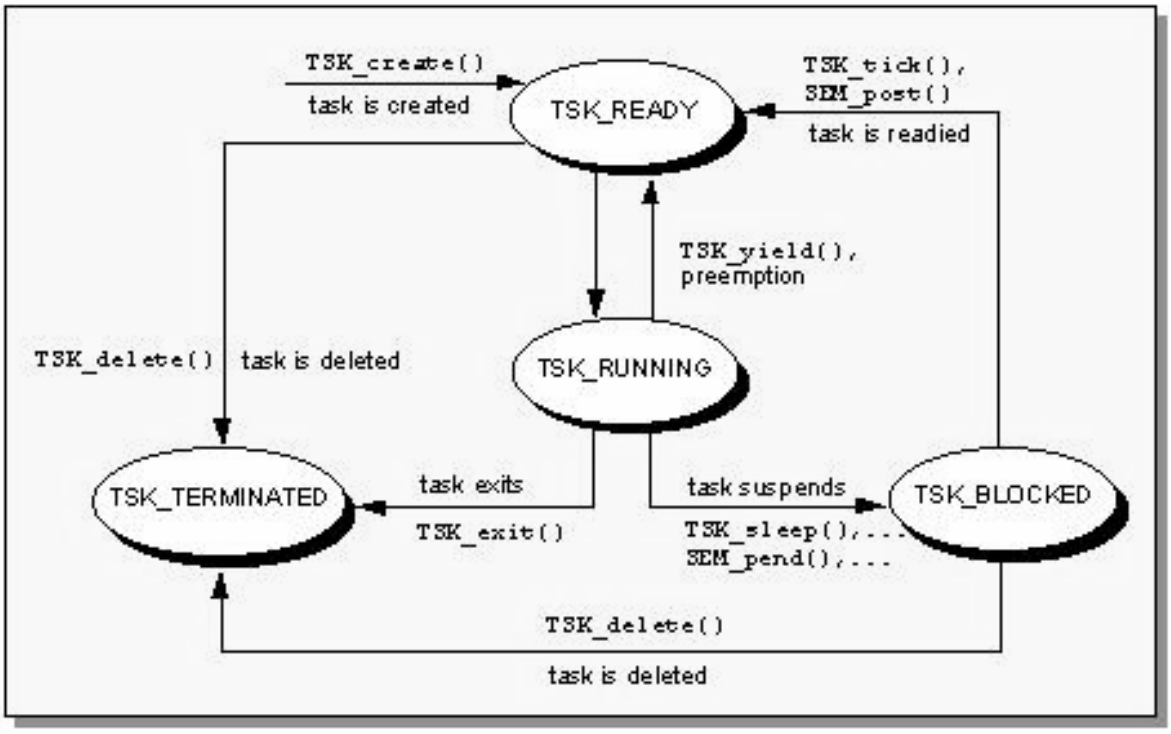

Figure 3. DSP/BIOS Synchronized Task States

Unlike HWI and SWI objects, tasks require their own configurable stack. This is an important point. The main advantage of SWI over TSK is that SWIs execute on a single stack. Designers leverage this into their designs by partitioning their application into multiple execution threads. In general, applications use SWIs for critical threads and TSKs for the normal multitasking threads. By partitioning to use both SWIs and TSKs, both the application and task stacks are smaller than if only one model is used.

DSP/BIOS tasks are very similar to task threads in other popular embedded real-time kernels such as PSOS™, VxWorks™, and Nucleus™. Developers familiar with the tasking paradigms of these kernels will find the DSP/BIOS kernel easy to grasp and familiar.

3.3 Semaphores Synchronize Events and Operations

In the DSP/BIOS kernel, the SEM module manages semaphores. The DSP/BIOS kernel implements counting semaphores. Counting semaphores keep an internal count of the number of corresponding resources available. When count is greater than zero, tasks do not block when acquiring a semaphore. This enables developers to implement a variety of synchronization schemes from mutual exclusion to event counting.

PSOS and VxWorks are trademarks of WINDRIVER Systems. Nucleus is a trademark of Advanced Technology Inc.

The semaphore count is initialized to *count* when it is created. In general, *count* is set to the number of resources that the semaphore is synchronizing. Semaphore operations include pending (decrementing the internal counter), and posting (incrementing the internal counter). `SEM_pend()` waits for a semaphore. `SEM_post()` is used to signal a semaphore. If the semaphore count is greater than 0, `SEM_pend()` simply decrements the count and returns. Otherwise, `SEM_pend()` waits for the semaphore to be posted by `SEM_post()`. If a task is waiting for the semaphore, `SEM_post()` removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, `SEM_post()` simply increments the semaphore count and returns.

DSP/BIOS semaphores include a time-out parameter to enable applications to select between waiting for a resource indefinitely before returning, waiting for a maximum amount of time, or returning immediately based on this value.

Tasks use semaphores to synchronize their execution. Typical examples are tasks waiting for data to become available from a peripheral device, or synchronizing access to shared memory or critical data with other tasks.

The DSP/BIOS kernel provides a dedicated kernel object to synchronize access to a shared resource. Resource locks, managed by the LCK module, arbitrate access to shared resources among several competing tasks. Lock objects are semaphores with an additional property. A task that currently owns the resource may request the resource lock any number of times without blocking. Under similar conditions, a regular semaphore may block the calling `Task` on the second call.

The MBX module manages mailboxes, which use an internal semaphore for access synchronization. Typically, mailboxes provide a utility to transfer messages or data between tasks that execute at different rates or priorities. Content of the mailbox is mostly user-defined to allow maximum flexibility in deployment. Tasks post messages to the mailbox or pend waiting for messages to arrive.

3.4 Data Transfers in the DSP/BIOS Kernel

Fundamental to all DSP applications is the need to acquire data, process it, and output the results. DSP applications typically process blocks of data at a time 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 data 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.

The DSP/BIOS kernel provides services to move blocks of data between peripheral devices and applications by two primary mechanisms: data streams and pipes. In both cases, the DSP/BIOS kernel does not copy data from source to destination — this is extremely inefficient. Rather, the DSP/BIOS kernel uses a buffer passing mechanism using address pointers that minimizes the amount of actual data being moved, and transfers the buffers without any copying.

3.4.1 Data Pipes Offer Universal I/O

The DSP/BIOS kernel introduces the data pipe mechanism through the PIP module. Data pipes are unidirectional and pass buffers of data, or frames, between any two DSP/BIOS execution threads. These threads are associated with hardware peripherals (via HWI), software interrupts, tasks, or the background Idle loop. Each data pipe specifically associates one reader and one writer function to access the data pipe.

Data pipes maintain their own memory pool, partitioned into a fixed-number of fixed-length buffers or *frames*. The data pipe *exchanges* these frames between a reader and the data pipe, and between the data pipe and the writer. Applications using data pipes must synchronize the requests for frames (either full or empty) with replacing them back to the pipe. For each frame request, the application must replace the frame back to the pipe before requesting another frame.

The configuration tool is required to create and configure data pipes in the DSP/BIOS kernel. The DSP/BIOS kernel does not supply runtime APIs to create data pipes during program execution.

3.4.2 Data Streams Provide Abstraction and Flexibility

The DSP/BIOS kernel introduces an alternative device-independent I/O model called data streams (see Figure 4). DSP/BIOS data streams transfer data between the application and any device through a consistent interface exposed through the streaming I/O module, SIO. Manipulations of the data streams include operations to open and close the streaming device, start, stop, and flush the stream, and to provide control.

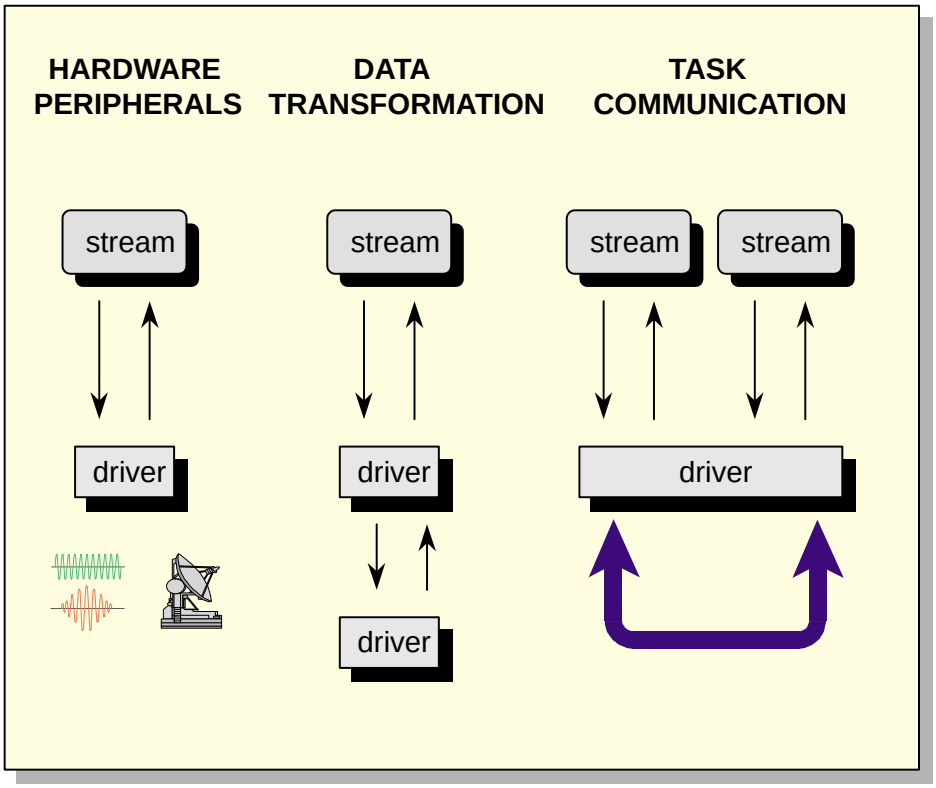

Figure 4. DSP/BIOS Streaming I/O Data Streams

In the DSP/BIOS kernel, 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. Applications make calls to `SIO_get()`(or `SIO_reclaim()`) and `SIO_put()` (or `SIO_issue()`), regardless of the underlying device.

Data streams offer flexible buffer management. Data buffers, or frames used by SIO may come from any source including SIO. SIO supports two modes of operation, a standard mode in which SIO creates the specified buffers, or an issue-reclaim mode in which the buffers are supplied to SIO by the application.

In addition, another feature of the data stream model implemented in SIO is selection between a synchronized frame exchange scheme and a scheme that allows the application to independently control issuing and reclaiming frame. In SIO, frames exchange between the calling task and the SIO module. If the SIO channel is reading data from a device, then the application will exchange an empty frame for a filled frame.

The standard SIO model (**SIO_STANDARD**), exchanges data frames, one at a time, with each call to `SIO_get()` or `SIO_put()`automatically. To obtain a new non-empty frame from the device, the application calls `SIO_get()` with one of the arguments being a pointer to an empty frame that the application has finished processing. SIO will exchange frames by assigning the frame pointer passed by the application to the non-empty frame. Meanwhile SIO will consume the empty frame in this exchange process. Important to note is that this exchange process does not copy any data. Through this exchange mechanism, the overhead associated with the exchange is constant, independent of frame size.

An alternative model (**SIO_ISSUERECLAIM**) allows the developer to independently control frame transfers. In the issue-reclaim model, the developer issues any number of frames (`SIO_issue()`) and reclaims them (`SIO_reclaim()`) independently under program control. In both models, SIO preserves the order of the frames exchanged by an internal set of queues, one for frames going to the device, and the other for frames coming from the device.

3.4.3 DEV Module Provides Device Independence to SIO

Complementing and interacting with SIO is the DEV module that manages the ***device-dependent*** drivers (see Figure 5). DEV provides a device-independent interface to SIO. SIO provides a device-independent interface to the DSP application.

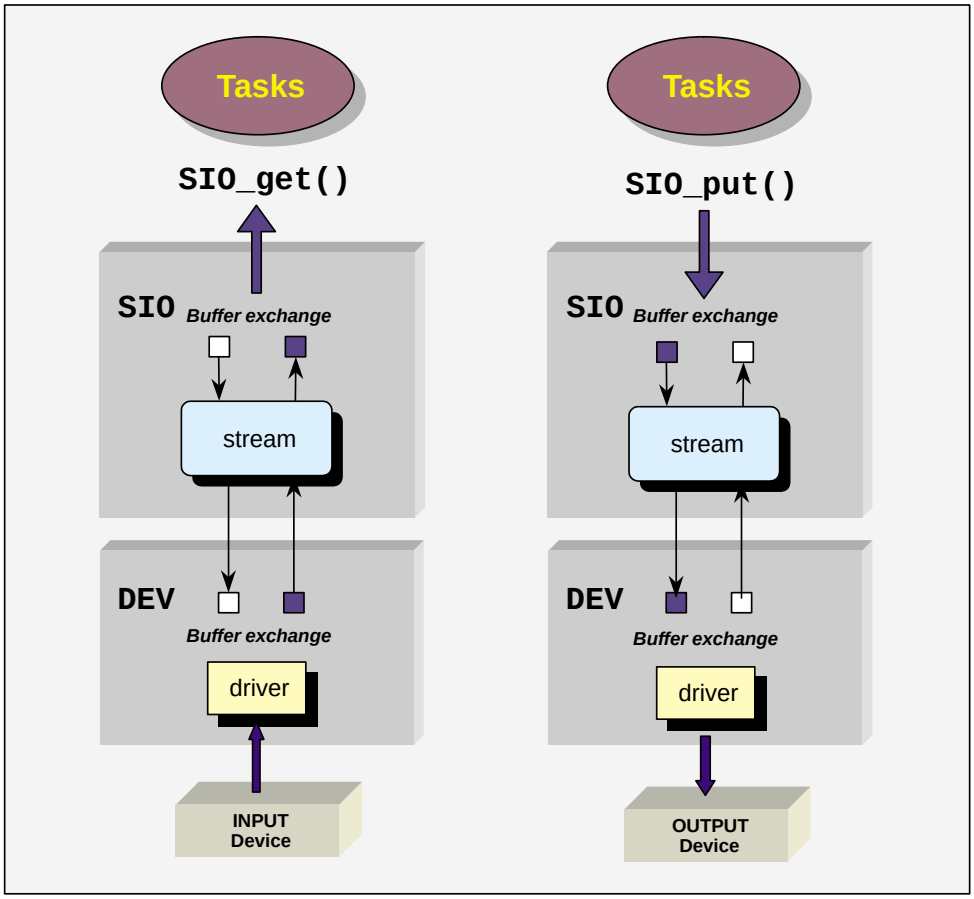

Figure 5. Streaming I/O Device-Dependent (DEV) and Independent (SIO) Modules

The underlying device driver performs a variety of functions including operations to open/close, start/stop, stream data, and initialize the driver. The DSP/BIOS kernel supports two classes of device drivers, terminating and stacking (see Figure 6).

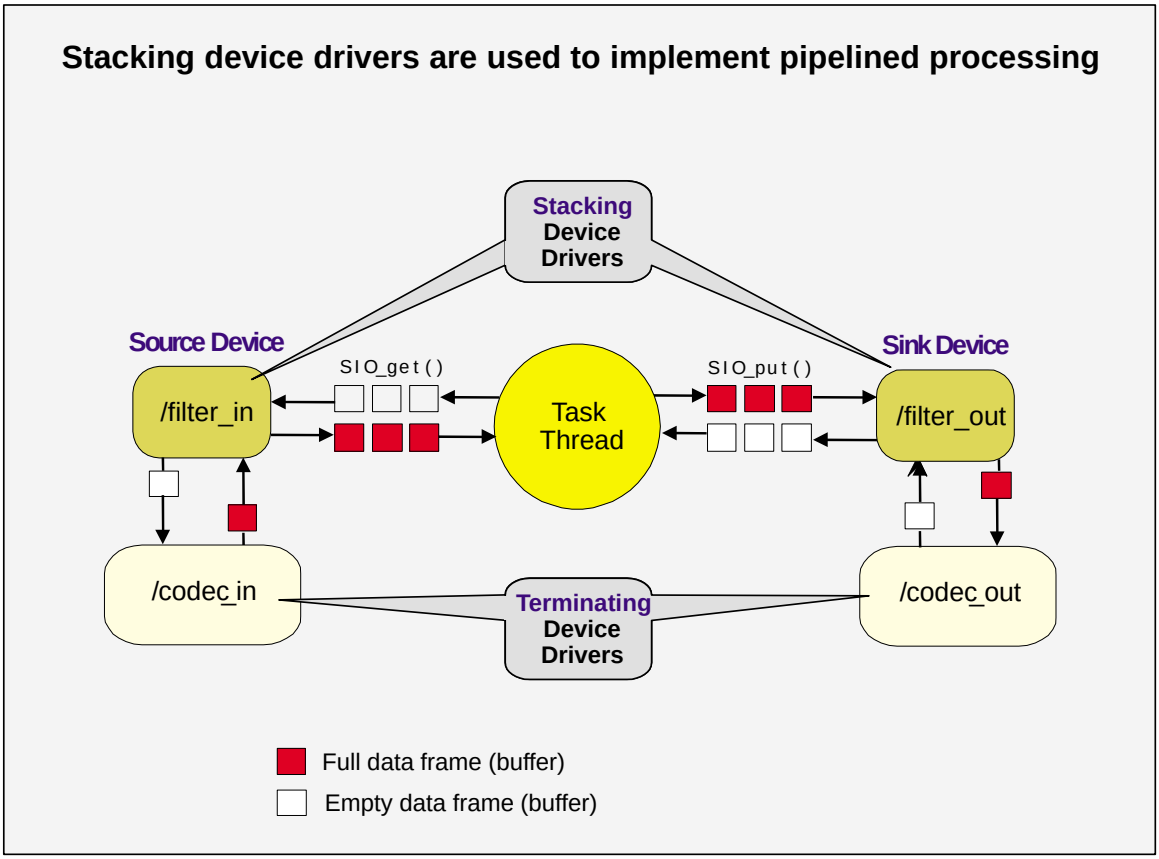

Figure 6. Stacking Device Drivers

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. Terminating device drivers supply SIO with either a data source or sink.

Stacking drivers are a special class of drivers that perform inline, pipelined processing. Stacking device drivers modify the data passing through them. A typical use of stacking device drivers is to implement operations such as scaling, filtering, or converting between data formats. The DSP/BIOS kernel supports two classes of stacking device drivers: in-place, and copying. Copying drivers allows the input and output buffers to have different sizes. This is useful for stacking devices that perform compression/decompression or packing and unpacking. If encapsulation and reuse of pipeline operations is beneficial, they become candidates for implementation as a stacking device driver. Developers create SIO channels specifying the terminating device driver and any stacking driver(s) to form the complete data path between the task and the device.

3.5 Creation and Configuration

A principle goal behind the creation of the DSP/BIOS kernel is to minimize resource utilization, both memory and CPU, by the system services themselves, freeing these valuable resources for applications. To achieve this, the DSP/BIOS kernel is by design both scalable and minimal in size. Scalability results from the modular construction of the DSP/BIOS runtime library and APIs. The DSP/BIOS scalable architecture allows binding only those modules that the application uses to minimize memory utilization. To further minimize resource utilization, the configuration tool allows creation and configuration of all DSP/BIOS objects at design time. This process is termed ***static creation and configuration***. Objects created in the configuration tool combine into an executable image and link with your application. Static creation and configuration minimizes DSP system resource requirements, and provides optimum performance.

If the program architecture (or framework), or availability of a kernel object or thread is required for the entire life of the program, then these elements require static declaration and configuration using the configuration tool.

Static creation does not limit the ability to design dynamic systems. Developers can implement dynamic systems with statically created objects. To accommodate this in static environments requires preallocating enough resources to handle all the possible requirements. For some systems, this can consume more resources than are actually required at any given time. Overlaying dynamic operation on a static infrastructure in complex systems is challenging, often leading to implementations that are unclear and difficult to maintain.

To achieve application flexibility easily, the DSP/BIOS kernel introduces ***dynamic creation and configuration***. The DSP/BIOS kernel permits the dynamic creation, deletion, and configuration of many kernel objects within an application during runtime (see Table 1). Using this paradigm, DSP/BIOS applications may create tasks to perform operations as needed, and delete them when finished. Similarly, dynamic creation and deletion of data paths between devices and Tasks is achievable. You can delete objects created dynamically. You cannot delete static objects. Only static objects declared using the configuration tool contain instrumentation and are visible to the real-time analysis tools within CCStudio. Dynamically created objects are not visible to the RTA tools. However all kernel object, static and dynamic, are visible to the Kernel Object Viewing CCStudio plug-in during application debug.

Typically, tradeoffs between application flexibility, performance, and resource utilization are required. The DSP/BIOS kernel provides developers a choice based on their application requirements.

3.5.1 Dynamic Memory Management

The DSP/BIOS kernel provides dynamic memory management APIs to allow applications to allocate and free memory during runtime. Developers define the memory segments to allocate from using the configuration tool and through runtime APIs. During runtime, the application makes API calls to the MEM module to allocate and free memory. Note that memory fragmenting may occur from frequent allocation and freeing of memory. To minimize memory footprint, the DSP/BIOS kernel does not provide defragmentation services or garbage collection. The developer must program the application to perform these services if they are required.

4 Conclusion

With the DSP/BIOS kernel, developers leverage a wealth of system services with the knowledge that Texas Instruments will ensure their availability in the future. These system services offer developers real time components to build their application on.

Advancements in DSP application complexities require more efficient application architectures that fully leverage the resources of DSP systems. This includes both memory and CPU requirements. Some applications are well suited for the static programming model of the DSP/BIOS kernel. Some will require dynamic capabilities to make optimal use of the resources, program performance, and cost. The DSP/BIOS kernel provides a mixture of services and capabilities that the developer can match against their application requirements. Combining the static, global nature of objects created through the configuration tool with the dynamic, localized nature of objects created within the program provides developers with the infrastructure to fully optimize their application's performance and resource utilization.

Building on DSP/BIOS modules and components, DSP developers can select which programming model is best suited to their application. Data streams using semaphore synchronization offer developers an alternative method for I/O that can include pipelining functions in the stream. Using stacking device drivers, pipelined operations on data flowing in the stream is configured on-the-fly, as dictated by the data flowing into the application.

5 References

1. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403).
2. *TMS320 DSP/BIOS User's Guide* (SPRU423).

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 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 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, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265