TMS320C6000 DSP/BIOS User’s Guide

Literature Number: SPRU303B
May 2000

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320C6000 DSP chips the ability to develop embedded real-time software. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

You should read and become familiar with the TMS320C6000 Application Programming Interface (API) Reference Guide (literature number SPRU403), the companion volume to this user’s guide.

Before you read this manual, you should follow the tutorials in the TMS320C6000 Code Composer Studio Tutorial (literature number SPRU301) to get an overview of DSP/BIOS. This manual discusses various aspects of DSP/BIOS in depth and assumes that you have at least a basic understanding of other aspects of DSP/BIOS.

Notational Conventions

This document uses the following conventions:

- The TMS320C6000 core is also referred to as ’C6000.
- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
Void copy(HST_Obj *input, HST_Obj *output)
{
    PIP_Obj     *in, *out;
    Uns         *src, *dst;
    Uns         size;
}
```

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Related Documentation From Texas Instruments

The following books describe the TMS320C6000 devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide (literature number SPRU403) describes the DSP/BIOS API functions, which are alphabetized by name. In addition, there are reference sections that describe the overall capabilities of each module, and appendices that provide tables that are useful to developers. The API Reference Guide is the companion to this user's guide.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C62x/C67x Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C62x/C67x DSPs and includes application program examples.

TMS320C62x/C67x CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62x/C67x CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, multichannel buffered serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x digital signal processors, development tools, and third-party support.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 and provides pinouts, electrical specifications, and timing for the device.

TMS320 DSP Designer’s Notebook: Volume 1 (literature number SPRT125) presents solutions to common design problems using ’C2x, ’C3x, ’C4x, ’C5x, and other TI DSPs.

TMS320C6000 Code Composer Studio Tutorial (literature number SPRU301) introduces the Code Composer Studio integrated development environment and software tools.

Related Documentation

You can use the following books to supplement this reference guide:

American National Standard for Information Systems-Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C)

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

Programming in C, Kochan, Steve G., Hayden Book Company

Trademarks

Contents

Chapter 1

About DSP/BIOS

DSP/BIOS gives developers of applications for DSP chips the ability to develop and analyze embedded real-time software. DSP/BIOS includes a small real-time library, an API for using real-time library services, and easy-to-use tools for configuration and for real-time program tracing and analysis.

Topic **Page**

1.1 DSP/BIOS Features and Benefits

DSP/BIOS and its plug-ins for Code Composer Studio are designed to minimize memory and CPU requirements on the target. This design goal was accomplished in the following ways:

- ❑ All DSP/BIOS objects can be created in the Configuration Tool and bound into an executable program image. This reduces code size and optimizes internal data structures.
- ❑ Instrumentation data (such as logs and traces) is formatted on the host.
- ❑ The API is modularized so that only the parts of the API that are used by the program need to be bound into the executable program.
- ❑ The library is optimized to require the smallest possible number of instruction cycles, with a significant portion implemented in assembly language.
- ❑ Communication between the target and the DSP/BIOS plug-ins is performed within the background idle loop. This ensures that the DSP/BIOS plug-ins do not interfere with the program's tasks. If the target CPU is too busy to perform background tasks, the DSP/BIOS plug-ins stop receiving information from the target until the CPU is available.

In addition, the DSP/BIOS API provides many powerful options for program development:

- ❑ A program can dynamically create and delete objects that are used in special situations. The same program can use both objects created dynamically and objects created with the Configuration Tool.
- ❑ The threading model provides thread types for a variety of situations. Hardware interrupts, software interrupts, tasks, idle functions, and periodic functions are all supported. You can control the priorities and blocking characteristics of threads through your choice of thread types.
- ❑ Structures to support communication and synchronization between threads are provided. These include semaphores, mailboxes, and resource locks.
- ❑ Two I/O models are supported for maximum flexibility and power. Pipes are used for target/host communication and to support simple cases in which one thread writes to the pipe and another reads from the pipe. Streams are used for more complex I/O and to support device drivers.
- ❑ Low-level system primitives are provided to make it easier to handle errors, create common data structures, and manage memory usage.

The DSP/BIOS API standardizes DSP programming for a number of TI chips, provides easy-to-use, powerful program development tools, and reduces the time required to create DSP programs in the following ways:

❏ The Configuration Tool generates code required to declare objects used within the program.

❏ The Configuration Tool detects errors earlier by validation properties before the program is even built.

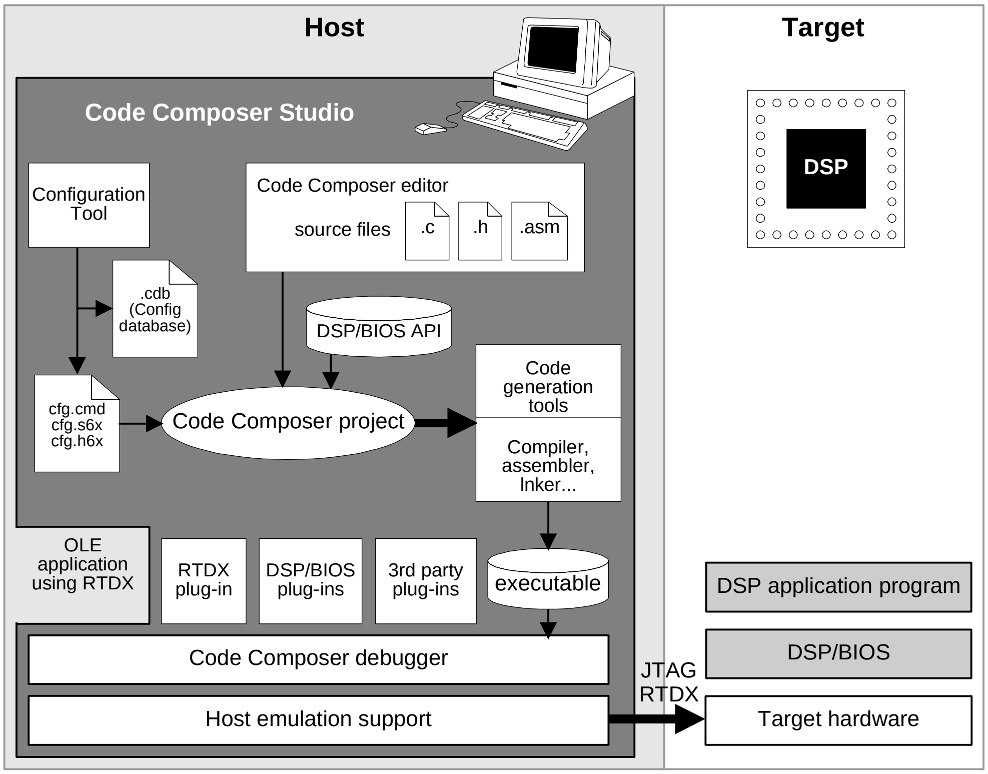
❏ Logging and statistics for DSP/BIOS objects are available at run-time without additional programming. Additional instrumentation can be programmed as needed.

❏ The DSP/BIOS plug-ins allow real-time monitoring of program behavior.

❏ DSP/BIOS provides a standard API. This allows DSP algorithm developers to provide code that can be more easily integrated with other program functions.

1.2 DSP/BIOS Components

This figure shows the components of DSP/BIOS within the program generation and debugging environment of Code Composer:



On the host PC, you write programs that use the DSP/BIOS API (in C or assembly). The Configuration Tool lets you define objects to be used in your program. You then compile or assemble and link the program. The DSP/BIOS plug-ins let you test the program on the target chip from Code Composer while monitoring CPU load, timing, logs, thread execution, and more. (The term *thread* is used to refer to any thread of execution, i.e., a hardware ISR, a software interrupt, a task, an idle function, or a periodic function.)

The following sections provide a brief overview of the DSP/BIOS components.

1.2.1 DSP/BIOS Real-Time Library and API

The small, firmware DSP/BIOS real-time library provides basic run-time services to embedded programs that run on the target hardware. It includes operations for scheduling threads, handling I/O, capturing information in real time, and more.

The DSP/BIOS API is divided into modules. Depending on what modules are configured and used by the application, the size of DSP/BIOS ranges from 200 to 2000 words of code. All the operations within a module begin with the letter codes shown here.

Module	**Description**
ATM	Atomic functions written in assembly language
C62	Target-specific functions
CLK	Clock manager
DEV	Device driver interface
GBL	Global setting manager
HST	Host channel manager
HWI	Hardware interrupt manager
IDL	Idle function manager
LCK	Resource lock manager
LOG	Event log manager
MBX	Mailbox manager
MEM	Memory section manager
PIP	Buffered pipe manager
PRD	Periodic function manager
QUE	Atomic Queue manager
RTDX	Real-Time Data Exchange settings
SEM	Semaphore manager
SIO	Stream I/O manager
STS	Statistics object manager
SWI	Software interrupt manager
SYS	System services manager
TRC	Trace manager
TSK	Multitasking manager

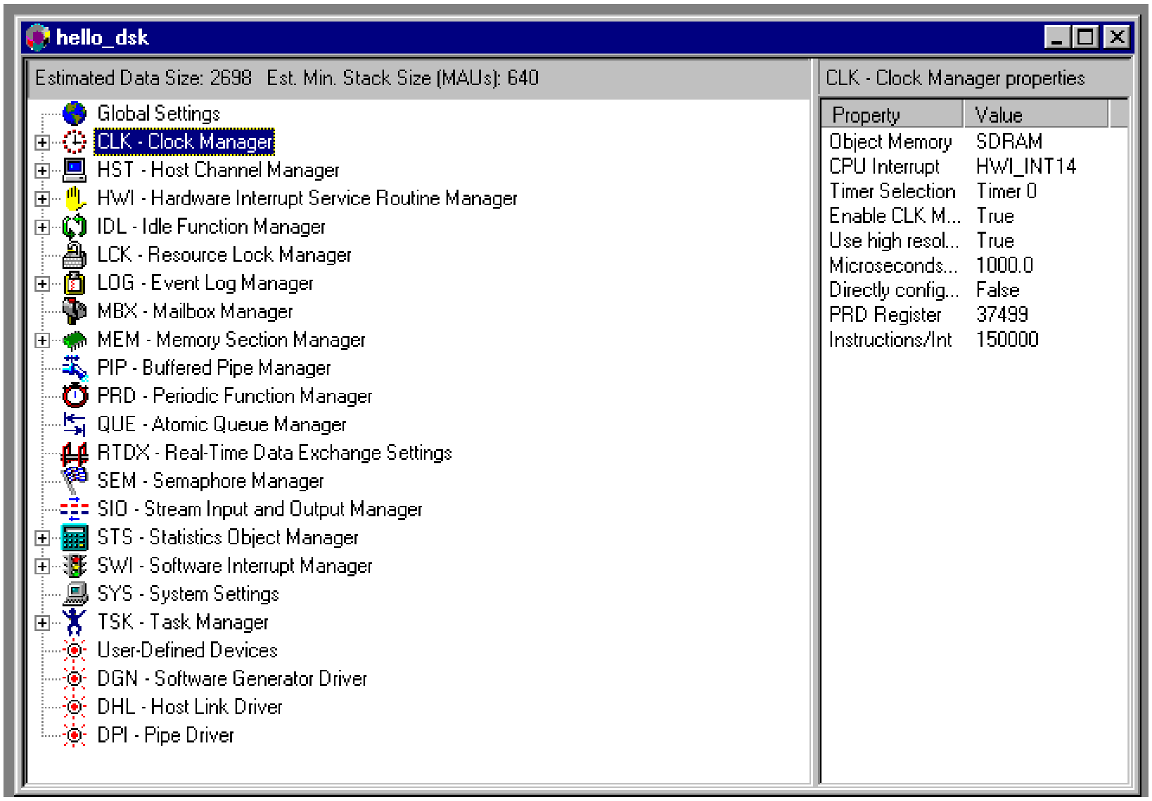
Application programs use DSP/BIOS by making calls to the API. For C programs, header files define the API. For applications that need assembly language optimization, an optimized set of macros is provided. Using C with DSP/BIOS is optional, as the real-time library itself is written in assembler to minimize time and space.

1.2.2 The DSP/BIOS Configuration Tool

The Configuration Tool has an interface similar to the Windows Explorer and has two roles:

❏ It lets you set a wide range of parameters used by the DSP/BIOS real-time library at run time.

❏ It serves as a visual editor for creating run-time objects that are used by the target application's DSP/BIOS API calls. These objects include software interrupts, tasks, I/O streams, and event logs. You also use this visual editor to set properties for these objects.



hello_dsk

Estimated Data Size: 2698 Est. Min. Stack Size (MAUs): 640

- Global Settings
- CLK - Clock Manager
- HST - Host Channel Manager
- HWI - Hardware Interrupt Service Routine Manager
- IDL - Idle Function Manager
- LCK - Resource Lock Manager
- LOG - Event Log Manager
- MBX - Mailbox Manager
- MEM - Memory Section Manager
- PIP - Buffered Pipe Manager
- PRD - Periodic Function Manager
- QUE - Atomic Queue Manager
- RTDX - Real-Time Data Exchange Settings
- SEM - Semaphore Manager
- SIO - Stream Input and Output Manager
- STS - Statistics Object Manager
- SWI - Software Interrupt Manager
- SYS - System Settings
- TSK - Task Manager
- User-Defined Devices
- DGN - Software Generator Driver
- DHL - Host Link Driver
- DPI - Pipe Driver

CLK - Clock Manager properties

Property	Value
Object Memory	SDRAM
CPU Interrupt	HWI_INT14
Timer Selection	Timer 0
Enable CLK M...	True
Use high resol...	True
Microseconds...	1000.0
Directly config...	False
PRD Register	37499
Instructions/Int	150000

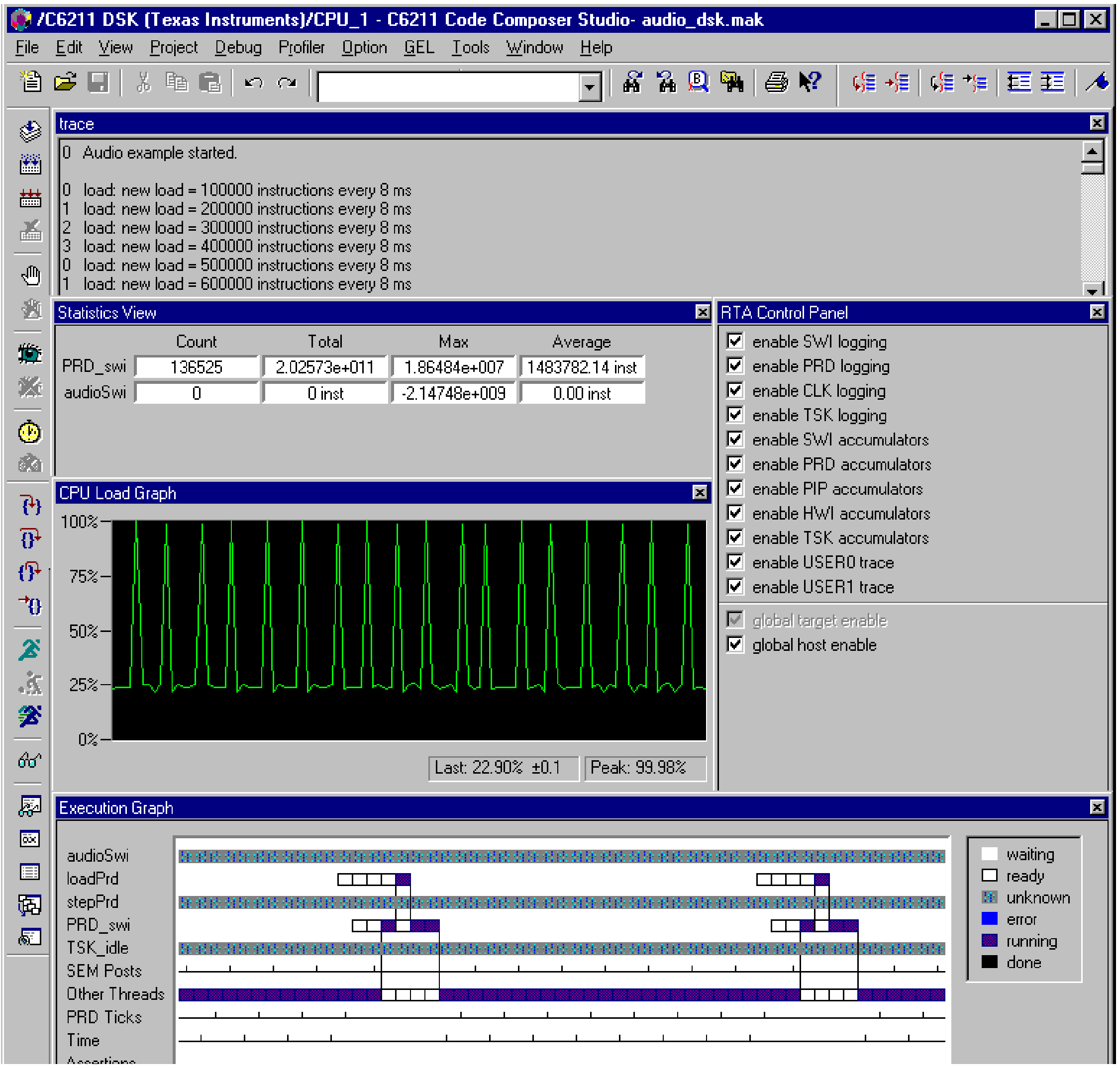
Using the Configuration Tool, DSP/BIOS objects can be pre-configured and bound into an executable program image. Alternately, a DSP/BIOS program can create and delete objects at run time. In addition to minimizing the target memory footprint by eliminating run-time code and optimizing internal data structures, creating static objects with the Configuration Tool detects errors earlier by validating object properties before program compilation.

The Configuration Tool generates files that link with code you write. See section 2.2, *Using the Configuration Tool*, page 2-3, for details.

1.2.3 The DSP/BIOS plug-ins

The DSP/BIOS plug-ins complement the Code Composer environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor a DSP application as it runs with minimal impact on the application's real-time performance.

Unlike traditional debugging, which is external to the executing program, program analysis requires that the target program contain real-time instrumentation services. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to the host through the DSP/BIOS plug-ins in Code Composer.


/C6211 DSK (Texas Instruments)/CPU_1 - C6211 Code Composer Studio- audio_dsk.mak
File Edit View Project Debug Profiler Option GEL Tools Window Help
trace
Statistics View
RTA Control Panel
CPU Load Graph
Execution Graph

```
0  Audio example started.

0  load: new load = 100000 instructions every 8 ms
1  load: new load = 200000 instructions every 8 ms
2  load: new load = 300000 instructions every 8 ms
3  load: new load = 400000 instructions every 8 ms
0  load: new load = 500000 instructions every 8 ms
1  load: new load = 600000 instructions every 8 ms
```

	Count	Total	Max	Average
PRD_swi	136525	2.02573e+011	1.86484e+007	1483782.14 inst
audioSwi	0	0 inst	-2.14748e+009	0.00 inst

- ☑ enable SWI logging
- ☑ enable PRD logging
- ☑ enable CLK logging
- ☑ enable TSK logging
- ☑ enable SWI accumulators
- ☑ enable PRD accumulators
- ☑ enable PIP accumulators
- ☑ enable HWI accumulators
- ☑ enable TSK accumulators
- ☑ enable USER0 trace
- ☑ enable USER1 trace
- ☑ global target enable
- ☑ global host enable

Last: 22.90% ±0.1 Peak: 99.98%

☐	waiting
□	ready
▨	unknown
■	error
▨	running
■	done

Several broad real-time program analysis capabilities are provided:

❑ **Program tracing**. Displaying events written to target logs, reflecting dynamic control flow during program execution

❑ **Performance monitoring**. Tracking summary statistics that reflect use of target resources, such as processor load and timing

❑ **File streaming**. Binding target-resident I/O objects to host files

When used in tandem with the other debugging capabilities of Code Composer, the DSP/BIOS real-time analysis tools provide critical views into target program behavior—where traditional debugging techniques that stop the target offer little insight—during program execution. Even after the debugger halts the program, information already captured by the host with the DSP/BIOS plug-ins can provide insights into the sequence of events that led up to the current point of execution.

Later in the software development cycle, when regular debugging techniques become ineffective for attacking problems arising from time-dependent interactions, the DSP/BIOS plug-ins have an expanded role as the software counterpart of the hardware logic analyzer.

1.3 Naming Conventions

Each DSP/BIOS module has a unique 3- or 4-letter name that is used as a prefix for operations (functions), header files, and objects for the module.

All identifiers beginning with upper-case letters followed by an underscore (XXX_*) should be treated as reserved words. Identifiers beginning with an underscore are also reserved for internal system names.

1.3.1 Module Header Names

Each DSP/BIOS module has two header files containing declarations of all constants, types, and functions made available through that module's interface.

❑ **xxx.h**. DSP/BIOS API header files for C programs. Your C source files should include std.h and the header files for any modules the C functions use.

❑ **xxx.h62**. DSP/BIOS API header files for assembly programs. Assembly source files should include the xxx.h62 header file for any module the assembly source uses. This file contains macro definitions specific to this chip. Data structure definitions shared for all supported chips are stored in the module.hti files, which are included by the xxx.h62 header files.

Your program must include the corresponding header for each module used in a particular program source file. In addition, C source files must include std.h before any module header files (see section 1.3.4, *Data Type Names*, page 1-11, for more information). The std.h file contains definitions for standard types and constants. Other than including std.h first, you may include the other header files in any sequence. For example:

```
#include <std.h>
#include <tsk.h>
#include <sem.h>
#include <prd.h>
#include <swi.h>
```

DSP/BIOS includes a number of modules that are used internally. These modules are undocumented and subject to change at any time. Header files for these internal modules are distributed as part of DSP/BIOS and must be present on your system when compiling and linking DSP/BIOS programs.

1.3.2 Object Names

System objects that are included in the configuration by default typically have names beginning with a 3- or 4-letter code for the module that defines or uses the object. For example, the default configuration includes a LOG object called LOG_system.

Note:

Objects you create with the Configuration Tool should use a common naming convention of your choosing. You might want to use the module name as a suffix in object names. For example, a TSK object that encodes data might be called encoderTsk.

1.3.3 Operation Names

The format for a DSP/BIOS API operation name is MOD_action where MOD is the letter code for the module that contains the operation, and action is the action performed by the operation. For example, the SWI_post function is defined by the SWI module; it posts a software interrupt.

This implementation of the DSP/BIOS API also includes several built-in functions that are run by various built-in objects. Here are some examples:

❏ **CLK_F_isr**. Run by the HWI_INT14 HWI object to provide the low-resolution CLK tick.

- **PRD_F_tick**. Run by the PRD_clock CLK object to provide the system tick.
- **PRD_F_swi**. Run by the highest priority SWI object, PRD_swi, to run the PRD functions.
- **_KNL_run**. Run by the lowest priority SWI object, KNL_swi, to run the task scheduler if it is enabled. This is a C function called KNL_run. An underscore is used as a prefix because the function is called from assembly code.
- **_IDL_loop**. Run by the lowest priority TSK object, TSK_idle, to run the IDL functions.
- **IDL_F_busy**. Run by the IDL_cpuLoad IDL object to compute the current CPU load.
- **RTA_F_dispatch**. Run by the RTA_dispatcher IDL object to gather real-time analysis data.
- **LNK_F_dataPump**. Run by the LNK_dataPump IDL object to transfer real-time analysis and HST channel data to the host.
- **HWI_unused**. Not actually a function name. This string is used in the Configuration Tool to mark unused HWI objects.

Note:

Your program code should not call any built-in functions whose names begin with MOD_F_. These functions are intended to be called only as function parameters specified within the Configuration Tool.

Symbol names beginning with MOD_ and MOD_F_ (where MOD is any letter code for a DSP/BIOS module) are reserved for internal use.

1.3.4 Data Type Names

The DSP/BIOS API does not explicitly use the fundamental types of C such as int or char. Instead, to ensure portability to other processors that support the DSP/BIOS API, DSP/BIOS defines its own standard data types. In most cases, the standard DSP/BIOS types are simply capitalized versions of the corresponding C types.

The following types are defined in the std.h header file:

Type	Description
Arg	Type capable of holding both Ptr and Int arguments

Type	Description
Bool	Boolean value
Char	Character value
Int	Signed integer value
LgInt	Large signed integer value
LgUns	Large unsigned integer value
Ptr	Generic pointer value
String	Zero-terminated (\0) sequence (array) of characters
Uns	Unsigned integer value
Void	Empty type

Additional data types are defined in std.h, but are not used by the DSP/BIOS API.

In addition, the standard constant NULL (0) is used by DSP/BIOS to signify an empty pointer value. The constants TRUE (1) and FALSE (0) are used for values of type Bool.

Object structures used by the DSP/BIOS API modules use a naming convention of MOD_Obj, where MOD is the letter code for the object's module. If your program uses any such objects, it should include an extern declaration for the object. For example:

```
extern LOG_Obj trace;
```

See , for more information.

1.3.5 Memory Segment Names

The memory segment names used by DSP/BIOS for the 'C6000 EVM are described in the following table.

Memory Segment	Description
IPRAM	Internal (on-chip) program memory
IDRAM	Internal (on-chip) data memory
SBSRAM	External SBSRAM on CE0
SDRAM0	External SDRAM on CE2

Memory Segment	Description
SDRAM1	External SDRAM on CE3

You can change the origin, size, and name of the default memory segments, with the exception of IPRAM and IDRAM, using the Configuration Tool.

The Configuration Tool defines standard memory sections and their default allocations as follows:

Memory Segment	Section
IDRAM	Application stack memory (.stack)
IDRAM	Application constants memory (.const)
IPRAM	Program memory (.text)
IDRAM	Data memory (.data)
IPRAM	Startup code memory (.sysinit)
IDRAM	C initialization records memory (.cinit)
IDRAM	Uninitialized variables memory (.bss)

You can change these default allocations by using the MEM manager in the Configuration Tool. See *MEM Module* in the *API Reference Guide*, for more details.

1.4 For More Information

For more information about the components of DSP/BIOS and the modules in the DSP/BIOS API, see the *TMS320C6000 Code Composer Studio Tutorial* and the *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*.

Chapter 2

Program Generation

This chapter describes the process of generating programs with DSP/BIOS. It also explains which files are generated by DSP/BIOS components and how they are used.

Topic **Page**

2.1 Development Cycle

DSP/BIOS supports iterative program development cycles. You can create the basic framework for an application and test it with a simulated processing load before the DSP algorithms are in place. You can easily change the priorities and types of program components that perform various functions.

A sample DSP/BIOS development cycle would include the following steps, though iteration could occur for any step or group of steps:

1) Write a framework for your program. You can use C or assembly code.
2) Use the Configuration Tool to create objects for your program to use.
3) Save the configuration file, which generates files to be included when you compile and link your program.
4) Compile and link the program using a makefile or a Code Composer project.
5) Test program behavior using a simulator or initial hardware and the DSP/BIOS plug-ins. You can monitor logs and traces, statistics objects, timing, software interrupts, and more.
6) Repeat steps 2-5 until the program runs correctly. You can add functionality and make changes to the basic program structure.
7) When production hardware is ready, modify the configuration file to support the production board and test your program on the board.

2.2 Using the Configuration Tool

The Configuration Tool is a visual editor with an interface similar to Windows Explorer. It allows you to initialize data structures and set various parameters of the DSP/BIOS Runtime. When you save a file, the Configuration Tool creates assembly and header files and a linker command file to match your settings. When you build your application, these files are linked with your application programs.

2.2.1 Creating a New Configuration

1) In Code Composer, open the Configuration Tool by selecting File→New→DSP/BIOS Config. Alternatively, you can open the Configuration Tool outside of Code Composer from the Start menu.
2) Select the appropriate template and click OK.
3) From the File menu, select New.
4) Double-click on the configuration template for the board you are using.

2.2.2 Creating a Custom Template

You can add a custom template or "seed file" by creating a configuration file and storing it in your include folder. This saves time by allowing you to define configuration settings for your hardware once and then reuse the file as a template.

For example, to build DSP/BIOS programs for the ’C67xx floating point DSP, you may use settings as provided (for the ’C62xx). Or you may instruct the Configuration Tool to create a new custom template file for projects that should take advantage of the floating point run-time library.

To create a custom template, perform the following steps:

1) Invoke the Configuration Tool from outside Code Composer via Start→Programs→Code Composer Studio ’C6000→Configuration Tool.
2) From the File menu, choose New.
3) In the New window select evm62.cdb and click OK.
4) Right-click on Global Settings and select Properties.
5) Set Target Board Name to evm67.
6) Set DSP Type to 6700 and click OK.
7) Select File→Save As. In the Save As dialog box navigate to ti\c6000\bios\include.
8) In the File Name box type evm67.cdb.
9) In the Save as type box select Seed files (*.cdb) and click Save.
10) In the Set Description Dialog type a description and click OK.
11) From the Configuration Tool's main menu select File→Exit.

2.2.3 Setting Global Properties for a Module

1) When you select a module (by clicking on it), the right side of the window shows the current properties for the module. (If you see a list of priorities instead of a property list, right-click on the module and select Property/value view. If the right side of the window is gray, this module has no global properties.)

   For help about a module, click and then click on the module.

2) Right-click the icon next to the module and select Properties from the pop-up menu. This opens the property sheet.

3) Change properties as needed. For help on the module's properties, click Help in the property sheet.

2.2.4 Creating Objects

Objects can be created using the Configuration Tool, which allows you to set the properties of each object, or dynamically by calling the function XXX_create(). This section describes objects created using the Configuration Tool. To create objects dynamically see Section 2.2.7, *Creating and Deleting Objects Dynamically*.

For typical DSP applications, most objects should be created with the DSP/BIOS Configuration Tool because they are used throughout program execution. A number of default objects are automatically defined in the configuration template. Creating objects with the Configuration Tool provides the following benefits:

❏ Improved access within DSP/BIOS plug-ins. The System Log shows the names of objects created with the Configuration Tool. In addition, you can view statistics only for objects created with the Configuration Tool.

❏ Reduced code size. For a typical module, the XXX_create() and XXX_delete() functions contain 50% of the code required to implement the module. If you avoid using any calls to TSK_create() and TSK_delete(), the underlying code for these functions is not included in the application program. The same is true for other modules. By creating objects with the Configuration Tool you can dramatically reduce the size of your application program.

Note:

SYS_printf() is probably the most memory intensive function in DSP/BIOS. Use the LOG functions instead of SYS_printf() to make your application smaller.

- ❏ Improved run-time performance. In addition to saving code space, avoiding dynamic creation of objects reduces the time your program spends performing system setup.

Creating objects with the Configuration Tool has the following limitations:

- ❏ Objects are created whether or not they are needed. You may want to create objects dynamically if they will be used only as a result of infrequent run-time events.
- ❏ You cannot delete objects created with the Configuration Tool at run-time using the XXX_delete functions.

Note:

No checks are performed to prevent an XXX_delete() function from being used on an object created with the Configuration Tool. If a program attempts to delete an object that was not created dynamically, SYS_error() is called.

2.2.5 Creating an Object Using the Configuration Tool

1) Right-click on a module and select Insert MOD, where MOD is the name of the module. This adds a new object for this module. (You cannot create an object for the GBL, HWI, SIO or SYS modules.)
2) Rename the object. Right-click on the name and choose Rename from the pop-up menu.
3) Right-click the icon next to the object and select Properties to open the property sheet.

Note:

When specifying C functions to be run by various objects, add an underscore before the C function name. For example, type _myfunc to run a C function called myfunc(). The underscore prefix is necessary because the Configuration Tool creates assembly source, and C calling conventions require an underscore before C functions called from assembly.

4) Change property settings and click OK. For help on specific properties, click Help in any property sheet.

2.2.6 Files Generated by the Configuration Tool

When you save a configuration file for your program with the Configuration Tool, the following files are created. These files are described in section 2.3, *Files Used to Create DSP/BIOS Programs*, page 2-7.

❏ program.cdb
❏ programcfg.h62
❏ programcfg.s62
❏ programcfg.cmd

2.2.7 Creating and Deleting Objects Dynamically

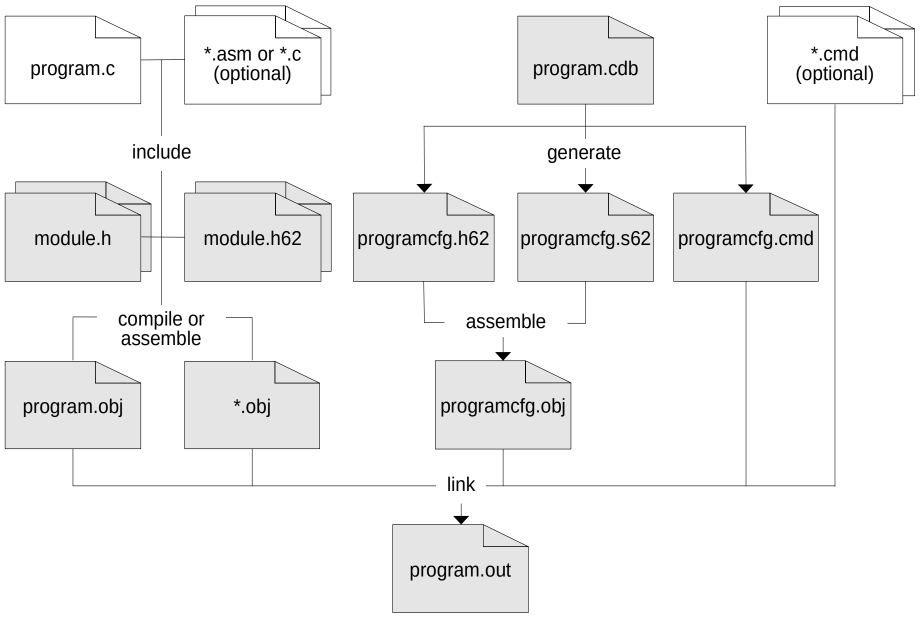
As illustrated by the TSK task example described previously, you can also create many, but not all, DSP/BIOS objects by calling the function XXX_create() where XXX names a specific module. Some objects can only be created in the Configuration Tool. Each XXX_create() function allocates memory for storing the object’s internal state information, and returns a handle used to reference the newly-created object when calling other functions provided by the XXX module.

Most XXX_create() functions accept as their last parameter a pointer to a structure of type XXX_Attrs used to assign attributes to the newly-created object. By convention, the object is assigned a set of default values if this parameter is NULL. These default values are contained in the constant structure XXX_ATTRS, enabling you to first initialize a variable of type XXX_Attrs and then selectively update its fields with application-dependent attribute values before calling XXX_create().

Objects created with XXX_create() are deleted by calling the function XXX_delete() which frees the object’s internal memory back to the system for later use.

2.3 Files Used to Create DSP/BIOS Programs

This diagram shows the files used to create DSP/BIOS programs. Files you write are represented with a white background; generated files are represented with a gray background.



A 62 in the file extension shown above indicates that the chip number abbreviation is used here. (For ’C6000 chips, the abbreviation is 62.)

❑ program.c. C program source file containing the main() function. You may also have additional .c source files and program .h files.

❑ *.asm. Optional assembly source file(s). One of these files can contain an assembly language function called _main as an alternative to using a C function called main().

❑ module.h. DSP/BIOS API header files for C programs. Your C source files should include std.h and the header files for any modules the C program uses.

❑ module.h62. DSP/BIOS API header files for assembly programs. Assembly source files should include the *.h62 header file for any module the assembly source uses.

❑ program.obj. Object file(s) compiled or assembled from your source file(s)

❑ *.obj. Object files for optional assembly source file(s)

- program.cdb. Configuration file, which stores configuration settings. This file is created by the Configuration Tool and used by both the Configuration Tool and the DSP/BIOS plug-ins.
- programcfg.h62. Header file generated by the Configuration Tool. This header file is included by the programcfg.s62 file.
- programcfg.s62. Assembly source generated by the Configuration Tool
- programcfg.cmd. Linker command file created by the Configuration Tool and used when linking the executable file. This file defines DSP/BIOS-specific link options and object names, and generic data sections for DSP programs (such as .text, .bss, .data, etc.).
- programcfg.obj. Object file created from source file generated by the Configuration Tool.
- *.cmd. Optional linker command file(s) that contains additional sections for your program not defined by the Configuration Tool.
- program.out. An executable program for the ’C6000 target (fully compiled, assembled, and linked). You can load and run this program with Code Composer.

2.3.1 Files Used by the DSP/BIOS Plugins

The following files are used by the DSP/BIOS plug-ins:

- program.cdb. The DSP/BIOS plug-ins use the configuration file to get object names and other program information.
- program.out. The DSP/BIOS plug-ins use the executable file to get symbol addresses and other program information.

2.4 Compiling and Linking Programs

You can build your DSP/BIOS executables using a Code Composer project or using your own makefile. Code Composer includes gmake.exe, GNU's make utility, and sample makefiles for gmake to build the tutorial examples.

2.4.1 Building with a Code Composer Project

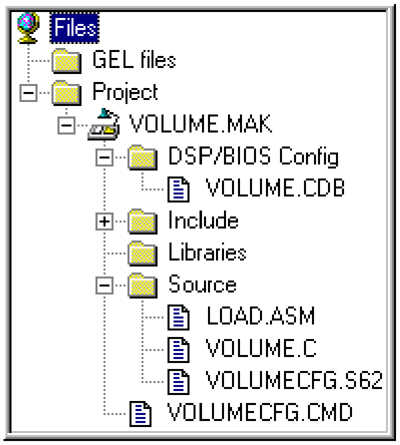
When building a DSP/BIOS application using a Code Composer project, you must add the following files to the project in addition to your own source code files:

❑ program.cdb (the configuration file)
❑ programcfg.cmd (the linker command file)

Code Composer adds programcfg.s62, the configuration source file, automatically.

Note that in a DSP/BIOS application, programcfg.cmd is your project's linker command file. programcfg.cmd already includes directives for the linker to search the appropriate libraries (e.g., bios.a62, rtdx.lib, rts6201.lib), so you do not need to add any of these library files to your project.

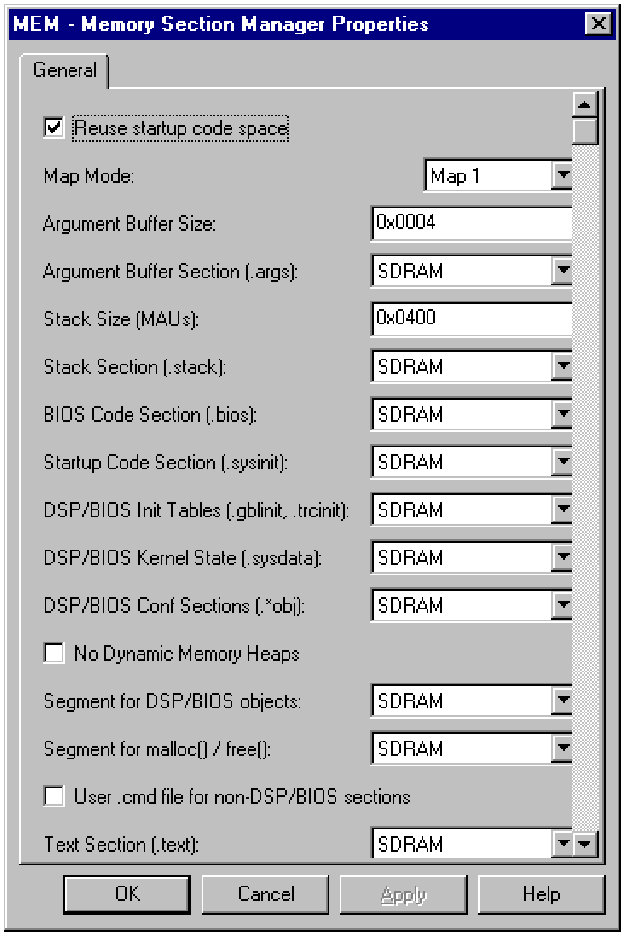
Code Composer automatically scans all dependencies in your project files, adding the necessary DSP/BIOS and RTDX header files for your configuration to your project's include folder.



For details on how to create a Code Composer project and build an executable from it, refer to the *Code Composer Studio User's Guide* or the *TMS320C6000 Code Composer Studio Tutorial*.

2.4.1.1 Building with Multiple Linker Command Files

For most DSP/BIOS applications the generated linker command file, programcfg.cmd, suffices to describe all memory sections and allocations. All DSP/BIOS memory sections and objects are handled by this linker command file. In addition, most commonly used sections (such as .text, .bss, .data, etc.) are already included in programcfg.cmd. Their locations (and sizes, when appropriate) can be controlled from the MEM manager in the Configuration Tool.


MEM - Memory Section Manager Properties

General

[✓] Reuse startup code space

Map Mode:	Map 1
Argument Buffer Size:	0x0004
Argument Buffer Section (.args):	SDRAM
Stack Size (MAUs):	0x0400
Stack Section (.stack):	SDRAM
BIOS Code Section (.bios):	SDRAM
Startup Code Section (.sysinit):	SDRAM
DSP/BIOS Init Tables (.gblinit, .trcinit):	SDRAM
DSP/BIOS Kernel State (.sysdata):	SDRAM
DSP/BIOS Conf Sections (.*obj):	SDRAM

[ ] No Dynamic Memory Heaps

Segment for DSP/BIOS objects:	SDRAM
Segment for malloc() / free():	SDRAM

[ ] User .cmd file for non-DSP/BIOS sections

Text Section (.text):	SDRAM

In some cases the application may require an additional linker command file (app.cmd) describing application-specific sections that are not described in the linker command file generated by the Configuration Tool (programcfg.cmd).

Note:

Code Composer allows only one linker command file per project. When both programcfg.cmd and app.cmd are required by the application, the project should use app.cmd (rather than programcfg.cmd) as the project's linker command file. To include programcfg.cmd in the linking process, you must add the following line to the beginning of app.cmd:

```
-lprogramcfg.cmd
```

Note that it is important that this line appear at the beginning, so that programcfg.cmd is the first linker command file used by the linker.

2.4.2 Makefiles

As an alternative to building your program as a Code Composer project, you can use a makefile.

In the following example, the C source code file is volume.c, additional assembly source is in load.asm, and the configuration file is volume.cdb. This makefile is for use with gmake, which is included with Code Composer. You can find documentation for gmake on the product CD in PDF format. Adobe Acrobat Reader is included. This makefile and the source and configuration files mentioned are located in the volume2 subdirectory of the tutorial directory of Code Composer.

A typical makefile for compiling and linking a DSP/BIOS program looks like the following.

```
# Makefile for creation of program named by the PROG variable
#
# The following naming conventions are used by this makefile:
#    <prog>.asm    - C62 assembly language source file
#    <prog>.obj    - C62 object file (compiled/assembled source)
#    <prog>.out    - C62 executable (fully linked program)
#    <prog>cfg.s62 - configuration assembly source file
#                    generated by Configuration Tool
#    <prog>cfg.h62 - configuration assembly header file
#                    generated by Configuration Tool
#    <prog>cfg.cmd - configuration linker command file
#                    generated by Configuration Tool
#

include $(TI_DIR)/c6000/bios/include/c62rules.mak

#
# Compiler, assembler, and linker options.
```

```
#
# -g enable symbolic debugging
CC62OPTS = -g
AS62OPTS =
# -q quiet run
LD62OPTS = -q

# Every BIOS program must be linked with:
#    $(PROG)cfg.o62 - object resulting from assembling
#                     $(PROG)cfg.s62
#    $(PROG)cfg.cmd - linker command file generated by
#                     the Configuration Tool. If additional
#                     linker command files exist,
#                     $(PROG)cfg.cmd must appear first.
#
PROG    = volume
OBJS    = $(PROG)cfg.obj load.obj
LIBS    =
CMDS    = $(PROG)cfg.cmd

#
# Targets:
#
all:: $(PROG).out

$(PROG).out: $(OBJS) $(CMDS)
$(PROG)cfg.obj: $(PROG)cfg.h62
$(PROG).obj:

$(PROG)cfg.s62 $(PROG)cfg.h62  $(PROG)cfg.cmd:
    @ echo Error: $@ must be manually regenerated:
    @ echo Open and save $(PROG).cdb within the BIOS Configuration Tool.
    @ check $@

.clean clean::
    @ echo removing generated configuration files ...
    @ remove -f $(PROG)cfg.s62 $(PROG)cfg.h62 $(PROG)cfg.cmd
    @ echo removing object files and binaries ...
    @ remove -f *.obj *.out *.lst *.map
```

You can copy an example makefile to your program folder and modify the makefile as necessary.

Unlike the Code Composer project, makefiles allow for multiple linker command files. If the application requires additional linker command files you can easily add them to the CMDS variable in the example makefile shown above. However, they must always appear after the programcfg.cmd linker command file generated by the Configuration Tool.

2.4.3 Referencing Pre-created DSP/BIOS Objects

Although DSP/BIOS itself is compiled using the small model, you may compile DSP/BIOS applications using either the ’C6000 compiler’s small model or any variation of the large model. (See the *TMS320C6000 Optimizing C Compiler User's Guide*.) In fact, you may mix compilation models within the application code provided the following conditions are met:

- ❏ Once the data page register (B14) is initialized to the start address of .bss at program startup, it must not be modified.
- ❏ All global data that is accessed by using a displacement relative to B14 must be placed no more than 32K bytes away from the beginning of the .bss section.

DSP/BIOS uses the .bss section to store global data. However, objects created with the Configuration Tool are not placed in the .bss section. This maximizes your flexibility in the placement of application data. For example, the frequently accessed .bss may be placed in on-chip memory while larger, less frequently accessed objects may be stored in external memory.

The small model makes assumptions about the placement of global data in order to reduce the number of instruction cycles. If you are using the small model (the default compilation mode) to optimize global data access, your code may need to be modified to insure that it references objects created with the Configuration Tool correctly.

There are four methods for dealing with this issue. These methods are described in the sections below and have the following pros and cons:

	Declare objects with far	**Use global object pointers**	**Objects adjacent to .bss**	**Compile with large model**
Code works independent of compilation model	Yes	Yes	Yes	Yes
Code works independent of object placement	Yes	Yes	No	Yes
C code is portable to other compilers	No	Yes	Yes	Yes
Size of all precreated objects not limited to 32K bytes	Yes	Yes	No	Yes
Minimizes size of .bss	Yes	Yes	No	Yes
Minimizes instruction cycles	No (3 cycles)	No (2-6 cycles)	Yes (1 cycle)	No (3 cycles)
Minimizes storage per object	No (12 bytes)	No (12 bytes)	Yes (4 bytes)	No (12 bytes)
Easy to remember when programming; easy to find errors	Somewhat	Error prone	Somewhat	Yes

2.4.3.1 Referencing Precreated Objects in the Small Model

In the small model, all compiled C code accesses global data relative to a data page pointer register. The register B14 is treated as a read-only register by the compiler and is initialized with the starting address of the .bss section during program startup. Global data is assumed to be at a constant offset from the beginning of the .bss section and this section is assumed to be at most 32K bytes in length. Global data can, therefore, be accessed with a single instruction like the following:

```
LDW  *+DP(_x), A0    ; load _x into A0 (DP = B14)
```

Since objects created with the Configuration Tool are not placed in the .bss section, you must ensure that application code compiled with the small model references them correctly. There are three ways to do this:

- ❑ Declare precreated objects with the far keyword. The TI compiler supports this common extension to the C language. The far keyword in a data declaration indicates that the data is not in the .bss section.

  For example, to reference a PIP object called inputObj that was created with the Configuration Tool, declare the object as follows:

  ```
  extern far PIP_Obj inputObj;
  if (PIP_getReaderNumFrames(&inputObj)) {
       . . .
  }
  ```

- ❑ Create and initialize a global object pointer. You may create a global variable that is initialized to the address of the object you want to reference. All references to the object must be made using this pointer, to avoid the need for the far keyword.

  ```
  extern PIP_Obj inputObj;
  PIP_Obj *input = &inputObj; /* input MUST be a global variable */
  if (PIP_getReaderNumFrames(input)) {
       . . .
  }
  ```

  Note that declaring and initializing the global pointer consumes an additional word of data (to hold the 32-bit address of the object).

  Also, if the pointer is a static or automatic variable this technique fails. The following code does not operate as expected when compiled using the small model:

  ```
  extern PIP_Obj inputObj;
  static PIP_Obj *input = &inputObj;   /* ERROR!!!! */
  if (PIP_getReaderNumFrames(input)) {
       . . .
  }
  ```

- ❑ Place all objects adjacent to .bss. If all objects are placed at the end of the .bss section and the combined length of the objects and the .bss data is less than 32K bytes, you can reference these objects as if they were allocated within the .bss section:

  ```
  extern PIP_Obj inputObj;
  if (PIP_getReaderNumFrames(&inputObj)) {
       . . .
  }
  ```

You can guarantee this placement of objects by using the Configuration Tool as follows:

a) Declare a new memory segment by inserting a MEM object with the MEM-Memory Section Manager and setting its properties (i.e., the base and length); or use one of the preexisting data memory MEM objects.

b) Place all objects that are referenced by small model code in this memory segment.

c) Place Uninitialized Variables Memory (.bss) in this same segment by right-clicking on the MEM manager and selecting Properties.

2.4.3.2 Referencing Precreated Objects in the Large Model

In the large model, all compiled C code accesses data by first loading the entire 32-bit address into an address register and then using the indirect addressing capabilities of the LDW instruction to load the data. For example:

```
MVKL     _x, A0     ; move low 16-bits of _x’s address into A0
MVKH    _x, A0     ; move high 16-bits of _x’s address into A0
LDW     *A0, A0    ; load _x into A0
```

Application code compiled with any of the large model variants is not affected by where precreated objects are located. If all code that directly references objects created with the Configuration Tool is compiled with any Large model option, code may reference the objects as ordinary data:

```
extern PIP_Obj inputObj;
if (PIP_getReaderNumFrames(&inputObj)) {
     . . .
}
```

Note that the -ml0 large model option is identical to small model except that all aggregate data is assumed to be far. This option causes all precreated objects to be assumed to be far objects but allows scalar types (such as int, char, long) to be accessed as near data. As a result, the performance degradation for many applications is quite modest.

2.5 DSP/BIOS Startup Sequence

When a DSP/BIOS application starts up, the startup sequence is determined by the calls in the boot.c file. A compiled version of this file is provided with the bios.a62 library. The DSP/BIOS startup sequence, as specified in the source file for boot.c is shown below. You should not need to make any changes to the boot.cfile, the source of which is listed beginning on page 2-18.

The steps followed in the startup sequence are:

1) Initialize the DSP. A DSP/BIOS program starts at the C environment entry point c_int00. The reset interrupt vector is set up to branch to c_int00 after reset. At the beginning of c_int00, the software stack pointer (B15), and the global page pointer (B14) are set up to point to the end of .stack and the beginning of .bss respectively. Control registers such as AMR, IER, and CSR are also initialized. Once the SP and DP are set up, the auto_init() routine is called to initialize the .bss section from the .cinit records.

2) Call BIOS_init to initialize the DSP/BIOS modules. BIOS_init is generated by the Configuration Tool and is located in the programcfg.s62 file. BIOS_init is responsible for basic module initialization. BIOS_init invokes the MOD_init macro for each DSP/BIOS module.

   - HWI_init sets up the ISTP and the interrupt selector registers, clears the IFR, and sets the NMIE bit in the IER. See the *API Reference Guide* for more information.

Note:

When configuring an interrupt with the Configuration Tool, DSP/BIOS plugs in the corresponding ISR (interrupt service routine) to the appropriate location of the interrupt service table. However, DSP/BIOS does not enable the interrupt bit in IER. It is your responsibility to do this at startup or whenever appropriate during the application execution.

HST_init initializes the host I/O channel interface. The specifics of this routine depend on the particular implementation used for the host to target link. For example, if RTDX is used, HST_init enables the bit in IER that corresponds to the hardware interrupt reserved for RTDX.

   - If the Auto calculate idle loop instruction count box was selected in the Idle Function Manager in the Configuration Tool, IDL_init calculates the idle loop instruction count at this point in the startup sequence. The idle loop instruction count is used to calibrate the CPU load displayed by the CPU Load Graph (see also section 3.5.2, *The CPU Load*, page 3-19).

3) Call your program's main routine. After all DSP/BIOS modules have completed their initialization procedures, your main routine is called. This routine can be written in assembly or C. Because the C compiler adds an underscore prefix to function names, this can be a C function called main or an assembly function called _main. The boot routine passes three parameters to main: argc, argv, and envp, which correspond to the C command line argument count, command line arguments array, and environment variables array.

   Since neither hardware or software interrupts are enabled yet, you can take care of initialization procedures for your own application (such as calling your own hardware initialization routines) from the main routine.

4) Call BIOS_start to start DSP/BIOS. Like BIOS_init, BIOS_start is also generated by the Configuration Tool and is located in the programcfg.s62 file. BIOS_start is called after the return from your main routine. BIOS_start is responsible for enabling the DSP/BIOS modules and invoking the MOD_startup macro for each DSP/BIOS module. For example:

   - CLK_startup sets up the PRD register, enables the bit in the IR for the timer selected in the CLK manager, and finally starts the timer. (This macro is only expanded if you enable the CLK manager in the Configuration Tool.)
   - PIP_startup calls the notifyWriter function for each created pipe object.
   - SWI_startup enables software interrupts.
   - TSK_startup enables taskings.
   - HWI_startup enables hardware interrupts by setting the GIE bit in the CSR.

5) Drop into the idle loop. By calling IDL_loop, the boot routine falls into the DSP/BIOS idle loop forever. At this point hardware and software interrupts can occur and preempt idle execution. Since the idle loop manages communication with the host, data transfer between the host and the target can now take place.

```
/*
 *  ======== boot.c ========
 *  BIOS Boot routine C6200.
 *
 *   Entry points:
 *       _c_int00:   called at reset.
 *
 */

/*******************************************************************************/
```

```
/* BOOT.C   v2.10 - Initialize the C60 C runtime environment                 */
/* Copyright (c) 1993-1998  Texas Instruments Incorporated                   */
/****************************************************************************/
extern void main(), _auto_init();

/*--------------------------------------------------------------------------*/
/* EXTERNAL BIOS SETUP FUNCTIONS                                            */
/*--------------------------------------------------------------------------*/
extern IDL_loop();
/*
 *  BIOS_init and BIOS_start are located in .sysinit rather than .text, hence
 *  they may require far branch.
 */
extern far void BIOS_init(), BIOS_start();

/*--------------------------------------------------------------------------*/
/* BIOS C ENVIROMENT SETUP SYMBOLS                                          */
/*--------------------------------------------------------------------------*/
asm("args: .sect  \".args\"");        /* address of arguments space */

/*--------------------------------------------------------------------------*/
/* ALLOCATE THE MEMORY FOR THE SYSTEM STACK.  THIS SECTION WILL BE SIZED     */
/* BY THE LINKER.                                                           */
/*--------------------------------------------------------------------------*/
asm(" .global __STACK_SIZE");
asm(" .global __stack");
asm("__stack: .usect   .stack, 0, 8");

/****************************************************************************/
/* C_INT00() - C ENVIRONMENT ENTRY POINT                                    */
/****************************************************************************/

/*
 * BIOS_reset is called via 'HWI_RESET' vector.  BIOS_reset is a duplicate
 * entry point to c_int00.  We use BIOS_reset to guarantee that the BIOS
 * startup code is linked from the bios library instead of from rts.lib.
 */
asm(" .sect \".sysinit\"");
asm(" .global BIOS_reset");
asm("BIOS_reset:");

/*
 * Put c_int00 in the .sysinit section.
 */
#pragma CODE_SECTION(c_int00,".sysinit")

extern void interrupt c_int00()
{
   /*-----------------------------------------------------------------------*/
   /* SET UP THE STACK POINTER IN B15.                                      */
   /* THE STACK POINTER POINTS 1 WORD PAST THE TOP OF THE STACK, SO SUBTRACT */
   /* 1 WORD FROM THE SIZE.                                                 */
   /*-----------------------------------------------------------------------*/
   asm("      mvkl __stack,SP");
   asm("      mvkh __stack,SP");
   asm("      mvkl __STACK_SIZE - 4,B0");
   asm("      mvkh __STACK_SIZE - 4,B0");
   asm("      add  B0,SP,SP");

   /*------------------------------------------------------------------------*/
   /* THE SP MUST BE ALIGNED ON AN 8-BYTE BOUNDARY.                          */
   /*------------------------------------------------------------------------*/
   asm("      and  ~7,SP,SP");

   /*------------------------------------------------------------------------*/
   /* SET UP THE GLOBAL PAGE POINTER IN B14.                                 */
   /*------------------------------------------------------------------------*/
   asm("      .global $bss");
   asm("      mvkl $bss,DP");
   asm("      mvkh $bss,DP");

   /*------------------------------------------------------------------------*/
   /* SET UP FLOATING POINT REGISTERS FOR C67 ONLY                           */
   /*------------------------------------------------------------------------*/
#ifdef _TMS320C6700
   asm("      mvk   0,B3");     /* round to nearest */
   asm("      mvc   B3,FADCR");
   asm("      mvc   B3,FMCR");
#endif

   /*------------------------------------------------------------------------*/
   /* INITIALIZE CONTROL REGISTERS (FOR BIOS ONLY)                           */
   /*------------------------------------------------------------------------*/
   asm("      mvk     0,B3");
   asm("      mvc     B3,AMR");          /* addressing mode register */
   asm("      mvc     B3,IER");          /* interrupt enable register */
   asm("      mvc     B3,CSR");          /* control status register */

   /*------------------------------------------------------------------------*/
   /* GET THE POINTER TO THE AUTOINITIALIZATION TABLES INTO THE FIRST        */
   /* ARGUMENT REGISTER (A4), AND CALL A FUNCTION TO PERFORM                 */
   /* AUTOINITIALIZATION.                                                    */
   /*------------------------------------------------------------------------*/
   asm("      .global cinit");
   asm("      mvkl cinit,A4");
   asm("      mvkh cinit,A4");

   /*------------------------------------------------------------------------*/
   /* PASS THE CURRENT DP TO THE AUTOINITIALIZATION ROUTINE.                 */
   /*------------------------------------------------------------------------*/
   asm("      mv   DP,B4");

   _auto_init();                          /* auto_init(A4, B4); */

   /*------------------------------------------------------------------------*/
   /* INITIALIZE THE RUNTIME ENVIRONMENT FOR BIOS                           */
   /*------------------------------------------------------------------------*/
   BIOS_init();

   asm("      mvkl args,A0");
asm("      mvkh args,A0");
asm("      ldw *+A0[2],A6");      /* envp */
asm("      ldw *+A0[1],B4");      /* argv */
asm("      ldw *+A0[0],A4");      /* argc */

/*------------------------------------------------------------------------*/
/* CALL THE USER’S PROGRAM.                                               */
/*------------------------------------------------------------------------*/
main();                               /* main(A4, B4, A6); */

/*------------------------------------------------------------------------*/
/* START RUNTIME FOR BIOS.                                                */
/*------------------------------------------------------------------------*/
BIOS_start();

/*------------------------------------------------------------------------*/
/* FALL INTO THE BIOS IDLE LOOP, NEVER RETURN.                            */
/*------------------------------------------------------------------------*/
IDL_loop();
 }
```

Chapter 3

Instrumentation

DSP/BIOS provides both explicit and implicit ways to perform real-time program analysis. These mechanisms are designed to have minimal impact on the application’s real-time performance.

Topic **Page**

3.1 Real-Time Analysis

Real-time analysis is the analysis of data acquired during real-time operation of a system. The intent is to easily determine whether the system is operating within its design constraints, is meeting its performance targets, and has room for further development.

The traditional debugging method for sequential software is to execute the program until an error occurs. You then stop the execution, examine the program state, insert breakpoints, and reexecute the program to collect information. This kind of cyclic debugging is effective for non-real-time sequential software. However, cyclic debugging is rarely as effective in real-time systems because real-time systems are characterized by continuous operation, nondeterministic execution, and stringent timing constraints.

The DSP/BIOS instrumentation APIs and the DSP/BIOS plug-ins are designed to complement cyclic debugging tools to enable you to monitor real-time systems as they run. This real-time monitoring data lets you view the real-time system operation so that you can effectively debug and performance-tune the system.

3.2 Software vs. Hardware Instrumentation

Software monitoring consists of instrumentation code that is part of the target application. This code is executed at run time, and data about the events of interest is stored in the target system’s memory. Thus, the instrumentation code uses both the computing power and memory of the target system.

The advantage of software instrumentation is that it is flexible and that no additional hardware is required. Unfortunately, because the instrumentation is part of the target application, performance and program behavior can be affected. Without using a hardware monitor, you face the problem of finding a balance between program perturbation and recording sufficient information. Limited instrumentation provides inadequate detail, but excessive instrumentation perturbs the measured system to an unacceptable degree.

DSP/BIOS provides a variety of mechanisms that allow you to precisely control the balance between intrusion and information gathered. In addition, the DSP/BIOS instrumentation operations all have fixed, short execution times. Since the overhead time is fixed, the effects of instrumentation are known in advance and can be factored out of measurements.

3.3 Instrumentation Performance Issues

When all implicit instrumentation is enabled, the CPU load increases less than 1 percent in a typical application. Several techniques have been used to minimize the impact of instrumentation on application performance:

- ❏ Instrumentation communication between the target and the host is performed in the background (IDL) thread, which has the lowest priority, so communicating instrumentation data does not affect the real-time behavior of the application.
- ❏ From the host, you can control the rate at which the host polls the target. You can stop all host interaction with the target if you want to eliminate all unnecessary external interaction with the target.
- ❏ The target does not store Execution Graph or implicit statistics information unless tracing is enabled. You also have the ability to enable or disable the explicit instrumentation of the application by using the TRC module and one of the reserved trace masks (TRC_USER0 and TRC_USER1).
- ❏ Log and statistics data are always formatted on the host. The average value for an STS object and the CPU load are computed on the host. Computations needed to display the Execution Graph are performed on the host.
- ❏ LOG, STS, and TRC module operations are very fast and execute in constant time, as shown in the following list:
  - ■ LOG_printf and LOG_event: approximately 32 instructions
  - ■ STS_add: approximately 18 instructions
  - ■ STS_delta: approximately 21 instructions
  - ■ TRC_enable and TRC_disable: approximately six instructions
- ❏ Each STS object uses only four words of data memory. This means that the host transfers only four words to upload data from a statistics object.
- ❏ Statistics are accumulated in 32-bit variables on the target and in 64-bit variables on the host. When the host polls the target for real-time statistics, it resets the variables on the target. This minimizes space requirements on the target while allowing you to keep statistics for long test runs.
- ❏ You can specify the buffer size for LOG objects. The buffer size affects the program’s data size and the time required to upload log data.
- ❏ For performance reasons, implicit hardware interrupt monitoring is disabled by default. When disabled, there is no effect on performance. When enabled, updating the data in statistics objects consumes between 20 and 30 instructions per interrupt for each interrupt monitored.

3.4 Instrumentation APIs

Effective instrumentation requires both operations that gather data and operations that control the gathering of data in response to program events. DSP/BIOS provides the following three API modules for data gathering:

❏ LOG (Event Log Manager). Log objects capture information about events in real time. System events are captured in the system log. You can create additional logs using the Configuration Tool. Your program can add messages to any log.

❏ STS (Statistics Object Manager). Statistics objects capture count, maximum, and total values for any variables in real time. Statistics about SWI (software interrupt), PRD (period), HWI (hardware interrupt), and PIP (pipe) objects can be captured automatically. In addition, your program can create statistics objects to capture other statistics.

❏ HST (Host Channel Manager). The host channel objects described in Chapter 7, *Input/Output Overview and Pipes*, allow a program to send raw data streams to the host for analysis.

LOG and STS provide an efficient way to capture subsets of a real-time sequence of events that occur at high frequencies or a statistical summary of data values that vary rapidly. The rate at which these events occur or values change may be so high that it is either not possible to transfer the entire sequence to the host (due to bandwidth limitations) or the overhead of transferring this sequence to the host would interfere with program operation. Therefore, DSP/BIOS also provides an API module for controlling the data gathering mechanisms provided by the other modules:

❏ TRC (Trace Manager). Controls which events and statistics are captured either in real time by the target program or interactively through the DSP/BIOS plug-ins.

Controlling data gathering is important because it allows you to limit the effects of instrumentation on program behavior, ensure that LOG and STS objects contain the necessary information, and start or stop recording of events and data values at run time.

3.4.1 Explicit vs. Implicit Instrumentation

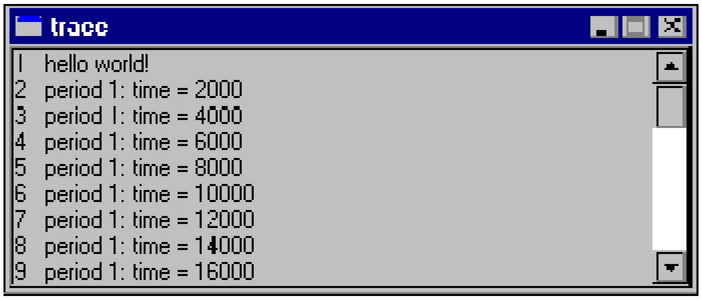
The instrumentation API operations are designed to be called explicitly by the application. The LOG module operations allow you to explicitly write messages to any log. The STS module operations allow you to store statistics about data variables or system performance. The TRC module allows you to enable or disable log and statistics tracing in response to a program event.

The LOG and STS APIs are also used internally by DSP/BIOS to collect information about program execution. These internal calls in DSP/BIOS routines provide implicit instrumentation support. As a result, even applications that do not contain any explicit calls to the DSP/BIOS instrumentation APIs can be monitored and analyzed using the DSP/BIOS plug-ins. For example, the execution of a software interrupt is recorded in a LOG object called LOG_system. In addition, worst-case ready-to-completion times for software interrupts and overall CPU load are accumulated in STS objects. The occurrence of a system tick can also be recorded in the Execution Graph. See section 3.4.4.2, *Control of Implicit Instrumentation*, page 3-16, for more information about what implicit instrumentation can be collected.

3.4.2 Event Log Manager (LOG Module)

This module manages LOG objects, which capture events in real time while the target program executes. You can use the Execution Graph, or create user-defined logs with the Configuration Tool.

User-defined logs contain any information your program stores in them using the LOG_event and LOG_printf operations. You can view messages in these logs in real time with the Event Log.



```
1  hello world!
2  period 1: time = 2000
3  period 1: time = 4000
4  period 1: time = 6000
5  period 1: time = 8000
6  period 1: time = 10000
7  period 1: time = 12000
8  period 1: time = 14000
9  period 1: time = 16000
```

The Execution Graph can also be viewed as a graph of the activity for each program component.

A log can be either fixed or circular. This distinction is valuable in applications that enable and disable logging programmatically (using the TRC module operations as described in section 3.4.4, *Trace Manager (TRC Module)*, page 3-13).

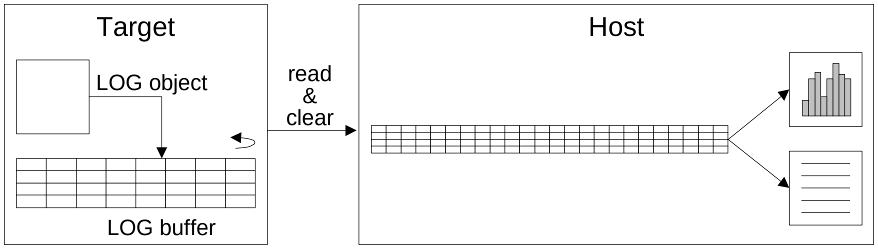
❑ Fixed. The log stores the first messages it receives and stops accepting messages when its message buffer is full. As a result, a fixed log stores the first events that occur since the log was enabled.

❑ Circular. The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.

You create LOG objects using the Configuration Tool, in which you assign properties such as the length and location of the message buffer.

You can specify the length of each message buffer in words. Individual messages use four words of storage in the log's buffer. The first word holds a sequence number. The remaining three words of the message structure hold event-dependent codes and data values supplied as parameters to operations such as LOG_event, which appends new events to a LOG object.

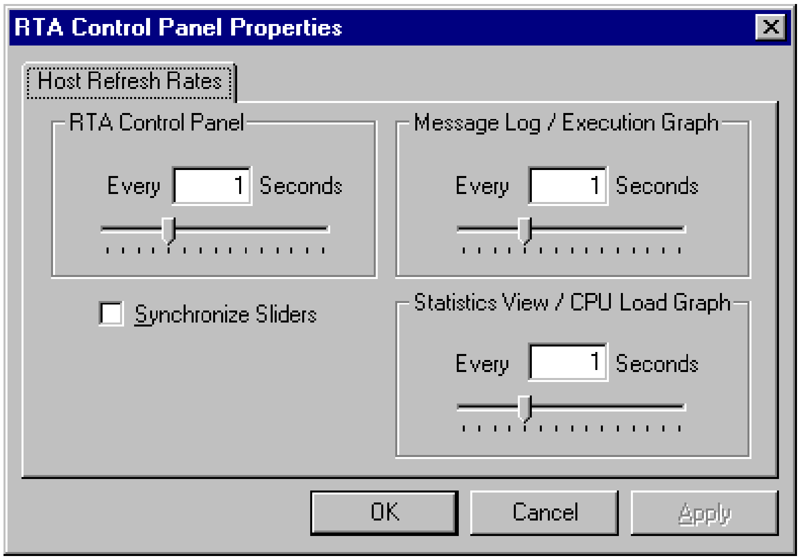
As shown in the following figure, LOG buffers are read from the target and stored in a much larger buffer on the host. Records are marked empty as they are copied up to the host.



LOG_printf uses the fourth word of the message structure for the offset or address of the format string (e.g., %d, %d). The host uses this format string and the two remaining words to format the data for display. This minimizes both the time and code space used on the target since the actual printf operation (and the code to perform the operation) are handled on the host.

LOG_event and LOG_printf both operate on logs atomically. This allows ISRs and other threads of different priorities to write to the same log without having to worry about synchronization.

Using the RTA Control Panel Property Page for each message log, you can control how frequently the host polls the target for information on a particular log. Right-click on the RTA Control Panel and choose the Property Page to set the refresh rate. If you set the refresh rate to 0, the host does not poll the target for log information unless you right-click on a log window and choose Refresh Window from the pop-up menu. You can also use the pop-up menu to pause and resume polling for log information.



RTA Control Panel: Every 1 Seconds

Message Log / Execution Graph: Every 1 Seconds

Statistics View / CPU Load Graph: Every 1 Seconds

☐ Synchronize Sliders

Log messages shown in a message log window are numbered (in the left column of the trace window) to indicate the order in which the events occurred. These numbers are an increasing sequence starting at 0. If your log never fills up, you can use a smaller log size. If a circular log is not long enough or you do not poll the log often enough, you may miss some log entries that are overwritten before they are polled. In this case, you see gaps in the log message numbers. You may want to add an additional sequence number to the log messages to make it clear whether log entries are being missed.

The online help in the Configuration Tool describes LOG objects and their parameters. See *LOG Module* in the *API Reference Guide* for information on the LOG module API calls.

3.4.3 Statistics Object Manager (STS Module)

This module manages objects called statistics objects, which store key statistics while a program runs.

You create individual statistics objects using the Configuration Tool. Each STS object accumulates the following statistical information about an arbitrary 32-bit wide data series:

❑ Count. The number of values in an application-supplied data series

❑ Total. The arithmetic sum of the individual data values in this series

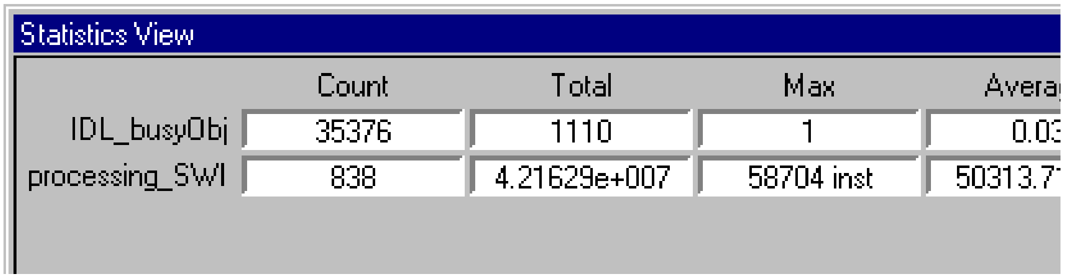
❑ Maximum. The largest value already encountered in this series

❑ Average. Using the count and total, the Statistics View plug-in also calculates the average

Calling the STS_add operation updates the statistics object of the data series being studied. For example, you might study the pitch and gain in a software interrupt analysis algorithm or the expected and actual error in a closed-loop control algorithm.

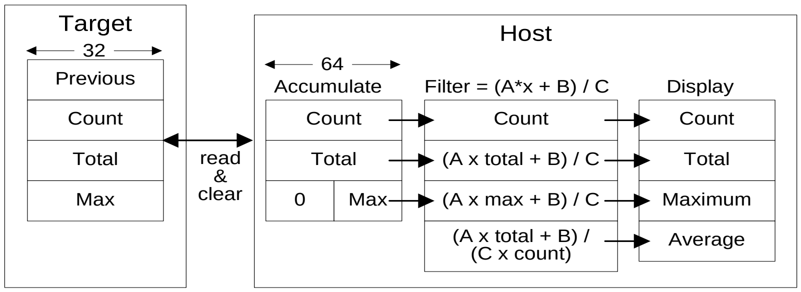
DSP/BIOS statistics objects are also useful for tracking absolute CPU use of various routines during execution. By bracketing appropriate sections of the program with the STS_set and STS_delta operations, you can gather real-time performance statistics about different portions of the application.

You can view these statistics in real time with the Statistics View.


Statistics View

	Count	Total	Max	Avera
IDL_busyObj	35376	1110	1	0.03
processing_SWI	838	4.21629e+007	58704 inst	50313.7

Although statistics are accumulated in 32-bit variables on the target, they are accumulated in 64-bit variables on the host. When the host polls the target for real-time statistics, it resets the variables on the target. This minimizes space requirements on the target while allowing you to keep statistics for long test runs. The Statistics View may optionally filter the data arithmetically before displaying it.



By clearing the values on the target, the host allows the values displayed to be much larger without risking lost data due to values on the target wrapping around to 0. If polling of STS data is disabled or very infrequent there is a possibility that the STS data wraps around, resulting in incorrect information.

While the host clears the values on the target automatically, you can clear the 64-bit objects stored on the host by right-clicking on the STS Data window and choosing Clear from the shortcut menu.

The host read and clear operations are performed atomically to allow any thread to update any STS object reliably. For example, an HWI function can call STS_add on an STS object and no data is missing from any STS fields.

This instrumentation process provides minimal intrusion into the target program. A call to STS_add requires approximately 18 instructions and an STS object uses only four words of data memory. Data filtering, formatting, and computation of the average is done on the host.

You can control the polling rate for statistics information with the RTA Control Panel Property Page. If you set the polling rate to 0, the host does not poll the target for information about the STS objects unless you right-click on the Statistics View window and choose Refresh Window from the pop-up menu.

3.4.3.1 Statistics About Varying Values

STS objects can be used to accumulate statistical information about a time series of 32-bit data values.

For example, let P_i be the pitch detected by an algorithm on the i^{th} frame of audio data. An STS object can store summary information about the time series $\{P_i\}$. The following code fragment includes the current pitch value in the series of values tracked by the STS object:

```
pitch = ‘do pitch detection‘
STS_add(&stsObj, pitch);
```

The Statistics View displays the number of values in the series, the maximum value, the total of all values in the series, and the average value.

3.4.3.2 Statistics About Time Periods

In any real-time system, there are important time periods. Since a period is the difference between successive time values, STS provides explicit support for these measurements.

For example, let T_i be the time taken by an algorithm to process the i^{th} frame of data. An STS object can store summary information about the time series $\{T_i\}$. The following code fragment illustrates the use of CLK_gethtime (high-resolution time), STS_set, and STS_delta to track statistical information about the time required to perform an algorithm:

```
STS_set(&stsObj, CLK_gethtime());
‘do algorithm‘
STS_delta(&stsObj, CLK_gethtime());
```

STS_set saves the value of CLK_gethtime as the previous value in the STS object. STS_delta subtracts this saved value from the value it is passed. The result is the difference between the time recorded before the algorithm started and after it was completed; i.e., the time it took to execute the algorithm (T_i). STS_delta then invokes STS_add and passes this result as the new value to be tracked.

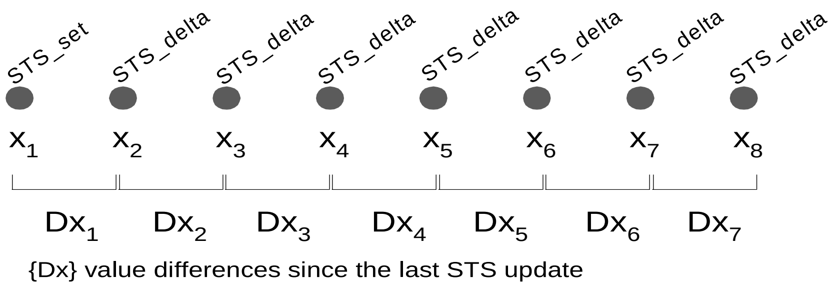
The host can display the count of times the algorithm was performed, the maximum time to perform the algorithm, the total time performing the algorithm, and the average time.

The previous field is the fourth component of an STS object. It is provided to support statistical analysis of a data series that consist of value differences, rather than absolute values.

3.4.3.3 Statistics About Value Differences

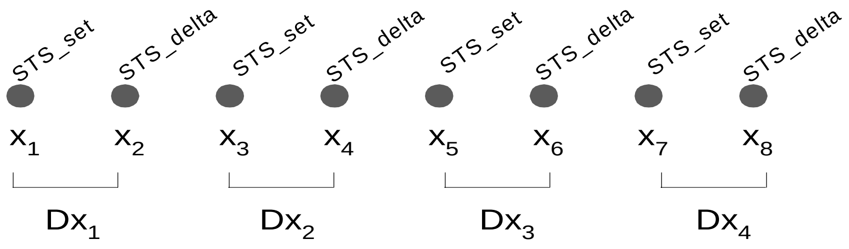
Both STS_set and STS_delta update the previous field in an STS object. Depending on the call sequence, you can measure specific value differences or the value difference since the last STS update. The following example gathers information about a difference between specific values.

```
STS_set(&sts, targetValue);
   "processing"
STS_delta(&sts, currentValue);
   "processing"
STS_delta(&sts, currentValue);
   "processing"
STS_delta(&sts, currentValue);
```


STS_set
STS_delta
STS_delta
STS_delta
STS_delta
STS_delta
STS_delta
STS_delta
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
Dx_1 Dx_2 Dx_3 Dx_4 Dx_5 Dx_6 Dx_7
{Dx} value differences since the last STS update

The next example gathers information about a value's difference from a base value.

```
STS_set(&sts, baseValue);
   "processing"
STS_delta(&sts, currentValue);
STS_set(&sts, baseValue);
   "processing"
STS_delta(&sts, currentValue);
```



{Dx} value differences with respect to a set base value

The online help in the Configuration Tool describes statistics objects and their parameters. See *STS Module* in the *API Reference Guide* for information on the STS module API calls.

3.4.4 Trace Manager (TRC Module)

The TRC module allows an application to enable and disable the acquisition of analysis data in real time. For example, the target can use the TRC module to stop or start the acquisition of data when it discovers an anomaly in the application's behavior.

Control of data gathering is important because it allows you to limit the effects of instrumentation on program behavior, ensure that LOG and STS objects contain the necessary information, and start or stop recording of events and data values at run time.

For example, by enabling instrumentation when an event occurs, you can use a fixed log to store the first n events after you enable the log. By disabling tracing when an event occurs, you can use a circular log to store the last n events before you disable the log.

3.4.4.1 Control of Explicit Instrumentation

You can use the TRC module to control explicit instrumentation as shown in this code fragment:

```
if (TRC_query(TRC_USER0) == 0) {
‘LOG or STS operation‘
}
```

Note:

TRC_query returns 0 if all trace types in the mask passed to it are enabled, and is not 0 if any trace types in the mask are disabled.

The overhead of this code fragment is just a few instruction cycles if the tested bit is not set. If an application can afford the extra program size required for the test and associated instrumentation calls, it is very practical to keep this code in the production application simplifying the development process and enabling field diagnostics. This is, in fact, the model used within DSP/BIOS itself.

3.4.4.2 Control of Implicit Instrumentation

The TRC module manages a set of trace bits that control the real-time capture of implicit instrumentation data through logs and statistics objects. For greater efficiency, the target does not store log or statistics information unless tracing is enabled. (You do not need to enable tracing for messages explicitly written with LOG_printf or LOG_event and statistics added with STS_add or STS_delta.)

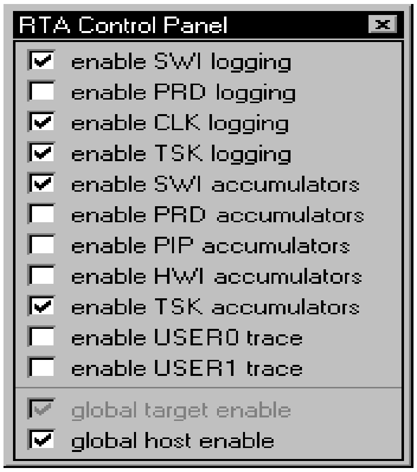
The trace bits allow the target application to control when to start and stop gathering system information. This can be important when trying to capture information about a specific event or combination of events.

DSP/BIOS defines the following constants for referencing specific trace bits:

Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Logs low-resolution clock interrupts	off
TRC_LOGPRD	Logs system ticks and start of periodic functions	off
TRC_LOGSWI	Logs posting, start, and completion of software interrupt functions	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes execution, and terminates	off
TRC_STSHWI	Gathers statistics on monitored register values within HWIs	off
TRC_STSPIP	Counts the number of frames read from or written to data pipe	off
TRC_STSPRD	Gathers statistics on the number of ticks elapsed during execution of periodic functions	off
TRC_STSSWI	Gathers statistics on number of instruction cycles or time elapsed from post to completion of software interrupt	off
TRC_STSTSK	Gather statistics on length of TSK execution	off
TRC_USER0 and TRC_USER1	Enables or disables sets of explicit instrumentation actions. You can use TRC_query to check the settings of these bits and either perform or omit instrumentation calls based on the result. DSP/BIOS does not use or set these bits.	off
TRC_GBLHOST	Simultaneously starts or stops gathering all enabled types of tracing. This bit must be set in order for any implicit instrumentation to be performed. This can be important if you are trying to correlate events of different types. This bit is usually set at run time on the host with the RTA Control Panel.	off
TRC_GBLTARG	This bit must also be set in order for any implicit instrumentation to be performed. This bit can only be set by the target program and is enabled by default.	on

You can enable and disable these trace bits in the following ways:

- ❑ From the host, use the RTA Control Panel. This window allows you to adjust the balance between information gathering and time intrusion at run time. By disabling various implicit instrumentation types, you lose information but reduce overhead of processing.

  
  RTA Control Panel

  - [✓] enable SWI logging
  - [ ] enable PRD logging
  - [✓] enable CLK logging
  - [✓] enable TSK logging
  - [✓] enable SWI accumulators
  - [ ] enable PRD accumulators
  - [ ] enable PIP accumulators
  - [ ] enable HWI accumulators
  - [✓] enable TSK accumulators
  - [ ] enable USER0 trace
  - [ ] enable USER1 trace
  - [✓] global target enable
  - [✓] global host enable

  You can control the refresh rate for trace state information by right-clicking on the Property Page of the RTA Control Panel. If you set the refresh rate to 0, the host does not poll the target for trace state information unless you right-click on the RTA Control Panel and choose Refresh Window from the pop-up menu.

- ❑ From the target code, enable and disable trace bits using the TRC_enable and TRC_disable operations, respectively. For example, the following C code would disable tracing of log information for software interrupts and periodic functions:

  ```
  TRC_disable(TRC_LOGSWI | TRC_LOGPRD);
  ```

  For example, in an overnight run you might be looking for a specific circumstance. When it occurs, your program can perform the following statement to turn off all tracing so that the current instrumentation information is preserved:

  ```
  TRC_disable(TRC_GBLTARG);
  ```

Any changes made by the target program to the trace bits are reflected in the RTA Control Panel. For example, you could cause the target program to disable the tracing of information when an event occurs. On the host, you can simply wait for the global target enable check box to be cleared and then examine the log.

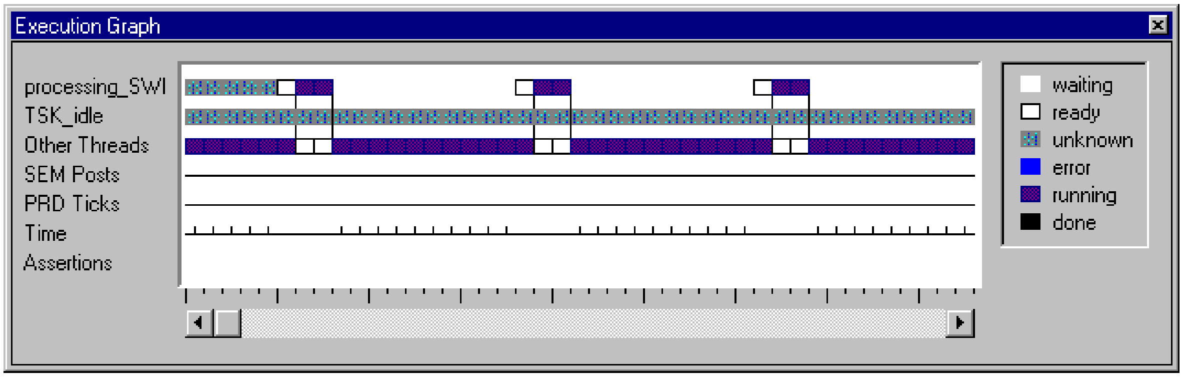
3.5 Implicit DSP/BIOS Instrumentation

The instrumentation needed to allow the DSP/BIOS plug-ins to display the Execution Graph, system statistics, and CPU load are all automatically built into a DSP/BIOS program to provide implicit instrumentation. You can enable different components of DSP/BIOS implicit instrumentation by using the RTA Control Panel plug-in in Code Composer, as described in section 3.4.4.2, *Control of Implicit Instrumentation*, page 3-15.

DSP/BIOS instrumentation is efficient—when all implicit instrumentation is enabled, the CPU load increases less than one percent for a typical application. See section 3.3, *Instrumentation Performance Issues*, page 3-3, for details about instrumentation performance.

3.5.1 The Execution Graph

The Execution Graph is a special log used to store information about SWI, PRD, and CLK processing. You can enable or disable logging for each of these object types at run time using the TRC module API or the RTA Control Panel in the host. The Execution Graph window in the host shows the Execution Graph information as a graph of the activity of each object.



CLK and PRD events are shown to provide a measure of time intervals within the Execution Graph. Rather than timestamping each log event, which is expensive (because of the time required to get the timestamp and the extra log space required), the Execution Graph simply records CLK events along with other system events.

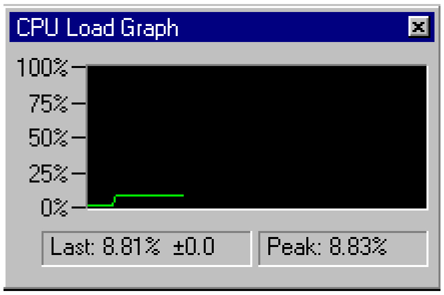
In addition to SWI, PRD, and CLK events, the Execution Graph shows additional information in the graphical display. Errors are indications that either a real-time deadline has been missed or an invalid state has been detected (either because the system log has been corrupted or the target has performed an illegal operation).

See section 4.1.5, *Yielding and Preemption*, page 4-7, for details on how to interpret the Execution Graph information in relation to DSP/BIOS program execution.

3.5.2 The CPU Load

The CPU load is defined as the percentage of instruction cycles that the CPU spends doing application work; i.e., the percentage of the total time that the CPU is:

- ❑ Running ISRs, software interrupts, or periodic functions
- ❑ Performing I/O with the host
- ❑ Running any user routine



When the CPU is not doing any of these, it is considered idle, even if the CPU is not in a power-save or hardware-idle mode.

All CPU activity is divided into work time and idle time. To measure the CPU load over a time interval T, you need to know how much time during that interval was spent doing application work (t_w) and how much of it was idle time (t_i). From this you can calculate the CPU load as follows:

$$\mathrm{CPUload} = \frac{t_w}{T} \times 100$$

Since the CPU is always either doing work or in idle it is represented as follows:

$$T = t_w + t_i$$

You can rewrite this equation:

$$\mathrm{CPUload} = \frac{t_w}{t_w + t_i} \times 100$$

You can also express CPU load using instruction cycles rather than time intervals:

$$\mathrm{CPUload} = \frac{c_w}{c_w + c_i} \times 100$$

In a DSP/BIOS application, the CPU is doing work when hardware interrupts are serviced, software interrupts and periodic functions are run, user functions are executed from the idle loop, HST channels are transferring data to the host, or real-time analysis information is being uploaded to the DSP/BIOS plug-ins. When the CPU is not performing any of those activities, it is going through the idle loop, executing the IDL_cpuLoad function, and calling the other DSP/BIOS IDL objects. In other words, the CPU idle time in a DSP/BIOS application is the time that the CPU spends doing the following routine.

To measure the CPU load in a DSP/BIOS application over a time interval T, it is sufficient to know how much time was spent going through the loop, shown in the IDL_loop example below, and how much time was spent doing application work; i.e., doing any other activity not included in the example:

```
'Idle_loop:
    Perform IDL_cpuLoad
     Perform all other IDL functions (these return without doing
        work)
    Goto IDL_loop'
```

Over a period of time T, a CPU with M MIPS (million instructions per second) executes M x T instruction cycles. Of those instruction cycles, c_w are spent doing application work. The rest are spent executing the idle loop shown above. If the number of instruction cycles required to execute this loop once is l_1, the total number of instruction cycles spent executing the loop is N x l_1 where N is the number of times the loop is repeated over the period T. Hence you have total instruction cycles equals work instruction cycles plus idle instruction cycles.

$$MT = c_w + Nl_1$$

From this expression you can rewrite c_w as:

$$c_w = MT - Nl_1$$

Using previous equations, you can calculate the CPU load in a DSP/BIOS application as:

$$CPUload = \frac{c_w}{MT} \times 100 = \frac{MT - NI_1}{MT} \times 100 = \left(1 - \frac{NI_1}{MT}\right) \times 100$$

To calculate the CPU load you need to know l_1 and the value of N for a chosen time interval T, over which the CPU load is being measured.

The IDL_cpuLoad object in the DSP/BIOS idle loop updates an STS object, IDL_busyObj, that keeps track of the number of times the IDL_loop runs, and the time as kept by the DSP/BIOS low-resolution clock (see section 4.8, *Timers, Interrupts, and the System Clock*, page 4-59). This information is used by the host to calculate the CPU load according to the equation above.

The host uploads the STS objects from the target at time intervals that you determine (the time interval between updates of the IDL_cpuLoad STS object, set in its Property Page). The information contained in IDL_busyObj is used to calculate the CPU load over the period of time between two uploads of IDL_busyObj. The IDL_busyObj count provides a measure of N (the number of times the idle loop ran). The IDL_busyObj maximum is not used in CPU load calculation. The IDL_busyObj total provides the value T in units of the low-resolution clock.

To calculate the CPU load you still need to know l_1 (the number of instruction cycles spent in the idle loop). When the Auto calculate idle loop instruction count box is checked in the Idle Function Manager in the Configuration Tool, DSP/BIOS calculates l_1 at initialization from BIOS_init.

The host uses the values described for N, T, l_1, and the CPU MIPS to calculate the CPU load as follows:

$$\mathrm{CPUload} = \left[1 - \frac{Nl_1}{MT}\right]100$$

Since the CPU load is calculated over the STS polling rate period, the value displayed is the average CPU load during that time. As the polling period increases, it becomes more likely that short spikes in the CPU load are not shown on the graph.

Considering the definition of idle time and work time used to calculate the CPU load, it follows that a DSP/BIOS application that performs only the loop shown in the previous IDL_loop example displays 0% CPU load. Since each IDL function runs once in every idle loop cycle, adding IDL objects to such an application dramatically increases the measure of CPU load. For example, adding an IDL function that consumes as many cycles as the rest of the components in the IDL_loop example results in a CPU load display of 50%. This increase in the CPU load is real, since the time spent executing the new IDL user function is, by definition, work time. However, this increase in CPU load does not affect the availability of the CPU to higher priority threads such as software or hardware interrupts.

In some cases you may want to consider one or more user IDL routines as CPU idle time, rather than CPU work time. This changes the CPU idle time to the time the CPU spends doing the following routine:

```
Idle_loop:
    Perform IDL_cpuLoad
    Perform user IDL function(s)
    Perform all other IDL functions
    Goto IDL_loop
```

The CPU load can now be calculated in the same way as previously described. However, what is considered idle time has now changed, and you need a new instruction cycle count for the idle loop described above. This new value must be provided to the host so that it can calculate the CPU load. To do this, enter the new cycle count in the Idle Function Manager Properties in the Configuration Tool. The IDL_loop cycle count can be calculated using the Code Composer Profiler to benchmark one pass through the IDL_loop when there is no host I/O or real-time analysis information transfer to the host, and the only routines executed are the IDL_cpuLoad function and any other user functions you want to include as idle time. (See the *TMS320C6000 Code Composer Studio Tutorial* manual for a description on how to use the Profiler.)

3.5.3 CPU Load Accuracy

The accuracy of the CPU load value measured as described above is affected by the accuracy in the measurements of T, N, and l_1.

❑ To measure T you use the low resolution clock, so you can only know T with the accuracy of the resolution of this clock. If the measured time is T ticks, but the real time elapsed is ticks + ε, with $0 < \varepsilon < 1$, the error introduced is as follows:

Error = | actual load – measured load |

$$= \left(1 - \frac{Nl_1}{M(T+\varepsilon)}\right) - \left(1 - \frac{Nl_1}{MT}\right)$$

$$= \frac{Nl_1}{M}\left(\frac{1}{T} - \frac{1}{T+\varepsilon}\right)$$

$$= \frac{Nl_1}{MT} \times \frac{\varepsilon}{T+\varepsilon}$$

Since (N x l_1)/(M x T) can be 1 at the most (when the CPU load is 0), the error is bounded:

$$\text{Error} \le \frac{\varepsilon}{T+\varepsilon} \qquad 0 < \varepsilon < 1$$

$$< \frac{1}{T}$$

In a typical application, the CPU load is measured at time intervals on the order of 1 second. The timer interrupt, which gives the resolution of the tick used to measure T, is triggered at time intervals on the order of 1 millisecond. Hence T is 1000 in low-resolution ticks. This results in a bounded error of less than 0.1% for the CPU load.

- To obtain a measurement of N, the host uses the integer value provided by the accumulated count in the IDL_busyObj STS object. However, the value of N could be overestimated or underestimated by as much as 1. The error introduced by this is:

  $$\begin{aligned} \text{Error} &= |\text{ actual load} - \text{measured load }| \\ &= \left(1 - \frac{Nl_1}{MT}\right) - \left(1 - \frac{(N+\varepsilon)l_1}{MT}\right) \\ &= (N + \varepsilon - N)\left(\frac{l_1}{MT}\right) \\ &= \frac{\varepsilon \times l_1}{MT} \qquad 0 < \varepsilon < 1 \\ &< \frac{l_1}{MT} \end{aligned}$$

  For a measurement period on the order of 1 second and a typical idle loop cycle count on the order of 200 instruction cycles, the additional error due to the approximation of N is far below the 0.1% error due to the resolution in the measure of T.

- Finally, there may also be an error in the calculation of l_1, the idle cycle instruction count, that affects the CPU load accuracy. This error depends on how l_1 is measured. When l_1 is autocalculated, BIOS_init uses the on-chip timer with CLKSRC = CPU/4 and the timer counter register value to estimate the idle loop instruction cycles count. Since the timer counter register increases at a rate of CPU/4 (one increase for every four CPU cycles), the resolution that can be achieved when measuring instruction cycles by reading the timer counter is worse than a single instruction cycle. This causes an uncertainty in the value estimated for l_1 that introduces a corresponding error in the value of the CPU load. This error is:

  $$\text{Error} = \Delta I_1\left(\frac{N}{MT}\right) \qquad \Delta I_1 = \text{error in the measured value of } l_1$$

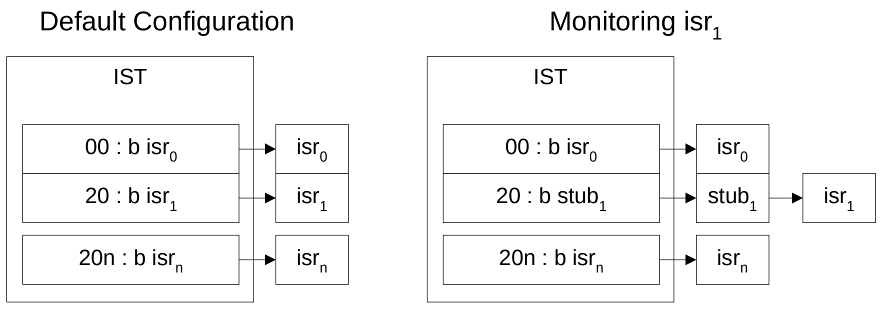
This error is the greatest when N is large; i.e., for CPU loads close to 0%. For this value, the error equals $\Delta l_1/l_1$; i.e., the maximum error in the CPU load calculation equals the percentage that Δl_1 represents in total idle cycle count l_1. Δl_1 is six instruction cycles when BIOS_init auto-calculates l_1 using the on-chip timer counter. Hence, the maximum CPU load error for a typical application with l_1 = 200 instruction cycles is 2.8% for a CPU load of 0.1%, and it decreases to less than 0.1% error for a CPU load of 99%, as shown in the following table.

CPU Load	CPU Load Error due to ΔI1
99%	<0.1%
80%	0.6%
75%	0.7%
50%	1.4%
25%	2.1%
10%	2.5%
1%	2.8%
0.1%	2.8%

The host calculates all the error components for each CPU load reported and displays the total accuracy for the CPU load value currently reported in the CPU load display. However, when you enter a value for l_1 manually, the last component of the error (due to l_1) is not added to the total error displayed.

3.5.4 Hardware Interrupt Count and Maximum Stack Depth

You can track the number of times an individual HWI function has been triggered by using the Configuration Tool to set the monitor parameter for an HWI object to monitor the stack pointer. An STS object is automatically created for each hardware ISR that is monitored.



For hardware interrupts that are not monitored, there is no overhead—control passes directly to the HWI function. For interrupts that are monitored, control first passes to a stub function generated by the Configuration Tool. This function reads the selected data location, passes the value to the selected STS operation, and finally branches to the HWI function.

The enable HWI objects check box in the RTA Control Panel must be selected in order for HWI function monitoring to take place. If this type of tracing is not enabled, the stub function branches to the HWI function without updating the STS object.

The number of times an interrupt is triggered is recorded in the Count field of the STS object. When the stack pointer is monitored, the maximum value reflects the maximum position of the top of the application stack when the interrupt occurs; this can be useful for determining the application stack size needed by an application. To determine the maximum depth of the stack, follow these steps:

1) Using the Configuration Tool right-click on the HWI object and select Properties. Change the monitor field to Stack Pointer and the operation field to STS_add(-*addr). The default setting for the type property can be left at "true". This gives you the minimum value of the stack pointer in the maximum field of the STS object. On the TMS320C6000 this is the top of the stack, since the stack grows downward in memory.

2) Link your program and use the nmti program described in Chapter 2, *Utility Programs* in the *API Reference Guide*, to find the address of the end of the application stack. Or, you can find the address of the end of the application stack in Code Composer by using a Memory window or

the map file to find the address referenced by the GBL_stackend symbol. (The GBL_stackbeg symbol references the top of the stack.)

3) Run your program and view the STS object that monitors the stack pointer for this HWI function in the Statistics View window.

4) Subtract the minimum value of the stack pointer (maximum field in the STS object) from the end of the application stack to find the maximum depth of the stack.

3.5.5 Monitoring Variables

In addition to counting hardware interrupt occurrences and monitoring the stack pointer, you can monitor any register or data value each time a hardware interrupt is triggered.

This implicit instrumentation can be enabled for any HWI object. Such monitoring is not enabled by default; the performance of your interrupt processing is not affected unless you enable this type of instrumentation in the Configuration Tool. The statistics object is updated each time hardware interrupt processing begins. Updating such a statistics object consumes between 20 and 30 instructions per interrupt for each interrupt monitored.

To enable implicit HWI instrumentation:

1) Open the properties window for any HWI object and choose a register to monitor in the monitor field.

   You can monitor any of the following values. When you choose a register or data value to monitor, the Configuration Tool automatically creates an STS object which stores statistics for any one of these values:

Data Value (type in addr field) Stack Pointer Top of SW Stack	a0 a1 a2 a3 a4 a5 a6 a7	a8 a9 a10 a11 a12 a13 a14 a15	b0 b1 b2 b3 b4 b5 b6 b7	b8 b9 b10 b11 b12 b13 b14 b15

2) Set the operation parameter to the STS operation you want to perform on this value.

   You can perform one of the following operations on the value stored in the data value or register you select. For all these operations, the count stores a count of the number of times this hardware interrupt has been executed. The max and total values are stored in the STS object on the target. The average is computed on the host.

STS Operation	Result
STS_add(*addr)	Stores maximum and total for the data value or register value
STS_delta(*addr)	Compares the data value or register value to the prev property of the STS object (or a value set consistently with STS_set) and stores the maximum and total differences.
STS_add(-*addr)	Negates the data value or register value and stores the maximum and total. As a result, the value stored as the maximum is the negated minimum value. The total and average are the negated total and average values.
STS_delta(-*addr)	Negates the data value or register value and compares the data value or register value to the prev property of the STS object (or a value set programmatically with STS_set). Stores the maximum and total differences. As a result, the value stored as the maximum is the negated minimum difference.
STS_add(|*addr|)	Takes the absolute value of the data value or register value and stores the maximum and total. As a result, the value stored as the maximum is the largest negative or positive value. The average is the average absolute value.
STS_delta(|*addr|)	Compares the absolute value of the register or data value to the prev property of the STS object (or a value set programmatically with STS_set). Stores the maximum and total differences. As a result, the value stored as the maximum is the largest negative or positive difference and the average is the average variation from the specified value.

3) You may also set the properties of the corresponding STS object to filter the values of this STS object on the host.

For example, you might want to watch the top of the software stack to see whether the application is exceeding the allocated stack size. The top of the software stack is initialized to 0xC0FFEE when the program is loaded. If this value ever changes, the application has either exceeded the allocated stack or some error has caused the application to overwrite the application’s stack.

One way to watch for this condition is to follow these steps:

1) In the Configuration Tool, enable implicit instrumentation on any regularly occurring HWI function. Right-click on the HWI object, select Properties, and change the monitor field to Top of SW Stack with STS_delta(*addr) as the operation.
2) Set the prev property of the corresponding STS object to 0xC0FFEE.
3) Load your program in Code Composer and use the Statistics View to view the STS object that monitors the stack pointer for this HWI function.
4) Run your program. Any change to the value at the top of the stack is seen as a non-zero total (or maximum) in the corresponding STS object.

3.5.6 Interrupt Latency

Interrupt latency is the maximum time between the triggering of an interrupt and when the first instruction of the ISR executes. You can measure interrupt latency for the timer interrupt by following these steps:

1) Configure the HWI object specified by the CPU Interrupt property of the CLK manager to monitor a Data Value.
2) Set the addr parameter to the address of the timer counter register for the on-chip timer device used by the CLK manager.
3) Set the type to unsigned.
4) Set the operation parameter to STS_add(*addr).
5) Set the Host Operation parameter of the corresponding STS object, HWI_INT14_STS, to A x X + B. Set A to 4 and B to 0.

The STS object HWI_INT14_STS then displays the maximum time (in instruction cycles) between when the timer interrupt was triggered and when the Timer Counter Register was able to be read. This is the interrupt latency experienced by the timer interrupt. The interrupt latency in the system is at least as large as this value.

3.6 Instrumentation for Field Testing

The embedded DSP/BIOS run-time library and DSP/BIOS plug-ins support a new generation of testing and diagnostic tools that interact with programs running on production systems. Since DSP/BIOS instrumentation is so efficient, your production program can retain explicit instrumentation for use with manufacturing tests and field diagnostic tools, which can be designed to interact with both implicit and explicit instrumentation.

3.7 Real-Time Data Exchange

Real-Time Data Exchange (RTDX) provides real-time, continuous visibility into the way DSP applications operate in the real world. RTDX allows system developers to transfer data between a host computer and DSP devices without interfering with the target application. The data can be analyzed and visualized on the host using any OLE automation client. This shortens development time by giving you a realistic representation of the way your system actually operates.

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 Code Composer Studio. Displays and analysis tools communicate with RTDX via an easy-to-use COM API to obtain the target data and/or to send data to the DSP application. Designers may use their choice of standard software display packages, including:

- ❏ National Instruments' LabVIEW
- ❏ Quinn-Curtis' Real-Time Graphics Tools
- ❏ Microsoft Excel

Alternatively, you can develop your own Visual Basic or Visual C++ applications. Instead of focusing on obtaining the data, you can concentrate on designing the display to visualize the data in the most meaningful way.

3.7.1 RTDX Applications

RTDX is well suited for a variety of control, servo, and audio applications. For example, wireless telecommunications manufacturers can capture the outputs of their vocoder algorithms to check the implementations of speech applications.

Embedded control systems also benefit from RTDX. Hard disk drive designers can test their applications without crashing the drive with improper signals to the servo-motor. Engine control designers can analyze changing factors (like heat and environmental conditions) while the control application is running.

For all of these applications, you can select visualization tools that display information in a way that is most meaningful to you. Future TI DSPs will increase RTDX bandwidth, providing greater system visibility to an even larger number of applications.

3.7.2 RTDX Usage

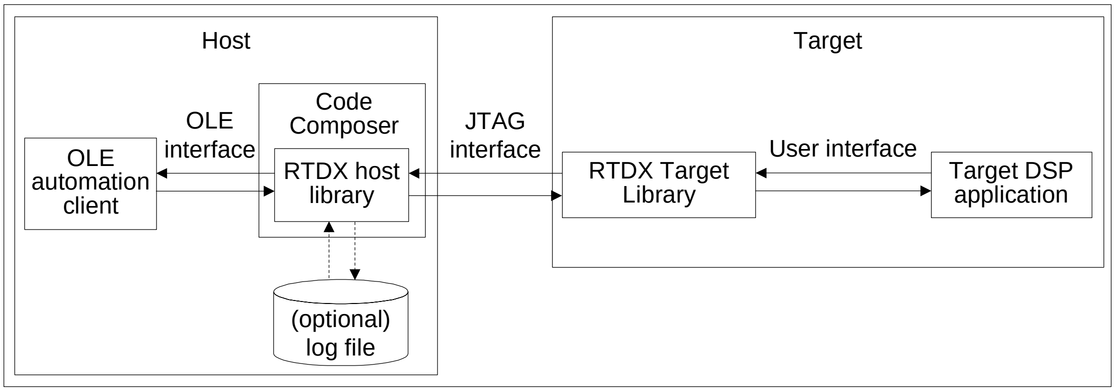
RTDX can be used within DSP/BIOS or, alternatively, without DSP/BIOS. The examples presented throughout the online help are written without DSP/BIOS.

RTDX is available with the PC-hosted Code Composer running Windows 95, Windows 98, or Windows NT version 4.0. RTDX is not currently available with the Code Composer Simulator.

This document assumes that the reader is familiar with C, Visual Basic or Visual C++, and OLE/ActiveX programming.

3.7.3 RTDX Flow of Data

Code Composer controls the flow of data between the host (PC) and the target (TI processor).



3.7.3.1 Target to Host Data Flow

To record data on the target, you must declare an output channel and write data to it using routines defined in the user interface. This data is immediately recorded into an RTDX target buffer defined in the RTDX target library. The data in the buffer is then sent to the host via the JTAG interface.

The RTDX host library receives this data from the JTAG interface and records it. The host records the data into either a memory buffer or to an RTDX log file (depending on the RTDX host recording mode specified).

The recorded data can be retrieved by any host application that is an OLE automation client. Some typical examples of OLE-capable host applications are:

❑ Visual Basic applications
❑ Visual C++ applications
❑ Lab View
❑ Microsoft Excel

Typically, an RTDX OLE automation client is a display that allows you to visualize the data in a meaningful way.

3.7.3.2 Host to Target Data Flow

For the target to receive data from the host, you must first declare an input channel and request data from it using routines defined in the user interface. The request for data is recorded into the RTDX target buffer and sent to the host via the JTAG interface.

An OLE automation client can send data to the target using the OLE Interface. All data to be sent to the target is written to a memory buffer within the RTDX host library. When the RTDX host library receives a read request from the target application, the data in the host buffer is sent to the target via the JTAG interface. The data is written to the requested location on the target in real time. The host notifies the RTDX target library when the operation is complete.

3.7.3.3 RTDX Target Library User Interface

The user interface provides the safest method of exchanging data between a target application and the RTDX host library.

The data types and functions defined in the user interface:

❑ Enable a target application to send data to the RTDX host library

❑ Enable a target application to request data from the RTDX host library

❑ Provide data buffering on the target. A copy of your data is stored in a target buffer prior to being sent to the host. This action helps ensure the integrity of the data and minimizes real-time interference.

❑ Provide interrupt safety. You can call the routines defined in the user interface from within interrupt handlers.

❑ Ensures correct utilization of the communication mechanism. It is a requirement that only one datum at a time can be exchanged between the host and target using the JTAG interface. The routines defined in the user interface handle the timing of calls into the lower-level interfaces.

3.7.3.4 RTDX Host OLE Interface

The OLE interface describes the methods that enable an OLE automation client to communicate with the RTDX host library.

The functions defined in the OLE interface:

❏ Enable an OLE automation client to access the data that was recorded in an RTDX log file or is being buffered by the RTDX Host Library

❏ Enable an OLE automation client to send data to the target via the RTDX host library

3.7.4 RTDX Modes

The RTDX host library provides the following modes of receiving data from a target application.

❏ Noncontinuous. In noncontinuous mode, data is written to a log file on the host.

  Noncontinuous mode should be used when you want to capture a finite amount of data and record it in a log file.

❏ Continuous. In continuous mode, the data is simply buffered by the RTDX host library; it is not written to a log file.

  Continuous mode should be used when you want to continuously obtain and display the data from a DSP application, and you don’t need to store the data in a log file.

Note:

To drain the buffer(s) and allow data to continuously flow up from the target, the OLE automation client must read from each target output channel on a continual basis. Failure to comply with this constraint may cause data flow from the target to cease, thus reducing the data rate, and possibly resulting in channels being unable to obtain data. In addition, the OLE automation client should open all target output channels on startup to avoid data loss to any of the channels.

3.7.5 Special Considerations When Writing Assembly Code

The RTDX functionality in the user library interface can be accessed by a target application written in assembly code.

See the Texas Instruments C compiler documentation for information about the C calling conventions, run-time environment, and runtime-support functions.

3.7.6 Target Buffer Size

The RTDX target buffer is used to temporarily store data that is waiting to be transferred to the host. You may want to reduce the size of the buffer if you are transferring only a small amount of data or you may need to increase the size of the buffer if you are transferring blocks of data larger than the default buffer size.

Using the Configuration Tool you can change the RTDX buffer size by right-clicking on the RTDX module and selecting Properties.

3.7.7 Sending Data From Target to Host or Host to Target

The user library interface provides the data types and functions for:

- ❏ Sending data from the target to the host
- ❏ Sending data from the host to the target

The following data types and functions are defined in the header file rtdx.h. They are available via DSP/BIOS or standalone.

- ❏ Declaration Macros
  - ■ RTDX_CreateInputChannel
  - ■ RTDX_CreateOutputChannel
- ❏ Functions
  - ■ RTDX_channelBusy
  - ■ RTDX_disableInput
  - ■ RTDX_disableOutput
  - ■ RTDX_enableOutput
  - ■ RTDX_enableInput
  - ■ RTDX_read
  - ■ RTDX_readNB
  - ■ RTDX_sizeofInput
  - ■ RTDX_write
- ❏ Macros
  - ■ RTDX_isInputEnabled
  - ■ RTDX_isOutputEnabled

  See *API Reference Guide* for detailed descriptions of all RTDX functions.

Chapter 4

Thread Scheduling

This chapter describes the types of threads a DSP/BIOS program can use, their behavior, and their priorities during program execution.

Topic **Page**

4.1 Overview

Many real-time DSP applications must perform a number of seemingly unrelated functions at the same time, often in response to external events such as the availability of data or the presence of a control signal. Both the functions performed and when they are performed are important.

These functions are called threads. Different systems define threads either narrowly or broadly. Within DSP/BIOS, the term is defined broadly to include any independent stream of instructions executed by the DSP. A thread is a single point of control that may contain a subroutine, an ISR, or a function call.

DSP/BIOS enables your applications to be structured as a collection of threads, each of which carries out a modularized function. Multithreaded programs run on a single processor by allowing higher-priority threads to preempt lower-priority threads and by allowing various types of interaction between threads, including blocking, communication, and synchronization.

Real-time application programs organized in such a modular fashion—as opposed to a single, centralized polling loop, for example—are easier to design, implement, and maintain.

DSP/BIOS provides support for several types of program threads with different priorities. Each thread type has different execution and preemption characteristics. The thread types (from highest to lowest priority) are:

❑ **Hardware interrupts (HWI):** includes CLK functions

❑ **Software interrupts (SWI):** includes PRD functions

❑ **Tasks (TSK)**

❑ **Background thread (IDL)**

These thread types are described briefly in the following section and discussed in more detail in the rest of this chapter.

4.1.1 Types of Threads

There are four major types of threads in a DSP/BIOS program:

❑ **Hardware interrupts (HWIs)**. Triggered in response to external asynchronous events that occur in the DSP environment. An HWI function (also called an interrupt service routine or ISR) is executed after a hardware interrupt is triggered in order to perform a critical task that is subject to a hard deadline. HWI functions are the threads with the highest priority in a DSP/BIOS application. HWIs should be used for application tasks that may need to run at frequencies approaching 200 kHz, and that need to be completed within deadlines of 2 to 100 microseconds. See section 4.2, *Hardware Interrupts*, page 4-11, for details about hardware interrupts.

- ❏ **Software interrupts (SWIs)**. Patterned after hardware ISRs. While ISRs are triggered by a hardware interrupt, software interrupts are triggered by calling SWI functions from the program. Software interrupts provide additional priority levels between hardware interrupts and the background thread. SWIs handle tasks subject to time constraints that preclude them from being run from the idle loop, but whose deadlines are not as severe as those of hardware ISRs. Like HWI's, SWI's threads always run to completion. Software interrupts should be used to schedule events with deadlines of 100 microseconds or more. SWIs allow HWIs to defer less critical processing to a lower-priority thread, minimizing the time the CPU spends inside an ISR, where other HWIs may be disabled. See section 4.3, *Software Interrupts*, page 4-18, for details about software interrupts.
- ❏ **Tasks (TSK).** Tasks have higher priority than the background thread and lower priority than software interrupts. Tasks differ from software interrupts in that they can be suspended during execution until necessary resources are available. DSP/BIOS provides a number of structures that can be use for inter-task communication and synchronization. These structures include queues, semaphores, and mailboxes. See section 4.4, *Tasks*, page 4-36, for details about tasks.
- ❏ **Background thread**. Executes the idle loop (IDL) at the lowest priority in a DSP/BIOS application. After main returns, a DSP/BIOS application calls the startup routine for each DSP/BIOS module and then falls into the idle loop. The idle loop is a continuous loop that calls all functions for the IDL objects. Each function must wait for all others to finish executing before it is called again. The idle loop runs continuously except when it is preempted by higher-priority threads. Only functions that do not have hard deadlines should be executed in the idle loop. See section 4.5, *The Idle Loop*, page 4-47, for details about the background thread.

There are several other kinds of functions that can be performed in a DSP/BIOS program. These are performed within the context of one of the thread types in the previous list.

- ❏ **Clock (CLK) functions**. Triggered at the rate of the on-chip timer interrupt. By default, these functions are triggered by the HWI_INT14 hardware interrupt and are performed as HWI functions. See section 4.8, *Timers, Interrupts, and the System Clock*, page 4-59, for details.
- ❏ **Periodic (PRD) functions**. Performed based on a multiple of either the on-chip timer interrupt or some other occurrence. Periodic functions are a special type of software interrupt. See section 4.9, *Periodic Function Manager (PRD) and the System Clock*, page 4-63, for details.
- ❏ **Data notification functions**. Performed when you use pipes (PIP) or host channels (HST) to transfer data. The functions are triggered when a frame of data is read or written to notify the writer or reader. These functions are performed as part of the context of the function that called PIP_alloc, PIP_get, PIP_free, or PIP_put.

4.1.2 Choosing Which Types of Threads to Use

The type and priority level you choose for each thread in an application program has an impact on whether the threads are scheduled on time and executed correctly. The Configuration Tool makes it easy to change a thread from one type to another.

Here are some rules for deciding which type of object to use for each task to be performed by a program:

❏ **SWI or TSK vs. HWI**. Perform only critical processing within hardware interrupt service routines. HWIs can run at frequencies approaching 200 kHz. Use software interrupts or tasks for events with deadlines around 100 microseconds or more. Your HWI functions should post software interrupts or tasks to perform lower-priority processing. Using lower-priority threads minimizes the length of time interrupts are disabled (interrupt latency), allowing other hardware interrupts to occur.

❏ **SWI vs. TSK.** Use software interrupts if functions have relatively simple interdependencies and data sharing requirements. Use tasks if the requirements are more complex. While higher-priority threads can preempt lower priority threads, only tasks can be suspended to wait for another event, such as resource availability. Tasks also have more options than SWIs when using shared data. All input needed by a software interrupt’s function should be ready when the program posts the SWI. The SWI object’s mailbox structure provides a way to determine when resources are available. SWIs are more memory efficient because they all run from a single stack.

❏ **IDL**. Create background functions to perform noncritical housekeeping tasks when no other processing is necessary. IDL functions do not typically have hard deadlines; instead they run whenever the system has unused processor time.

❏ **CLK**. Use CLK functions when you want a function to be triggered directly by a timer interrupt. These functions run as HWI functions and should take minimal processing time. The default CLK object, PRD_clock, causes a tick for the periodic functions. You can add additional CLK objects to run at the same rate. However, you should minimize the time required to perform all CLK functions because they run as HWI functions.

❏ **PRD**. Use PRD functions when you want a function to run at a rate based on a multiple of the on-chip timer’s low-resolution rate or another event (such as an external interrupt). These functions run as SWI functions.

❏ **PRD vs. SWI**. All PRD functions run at the same SWI priority, so one PRD function cannot preempt another. However, PRD functions can post lower-priority software interrupts for lengthy processing routines. This ensures that the PRD_swi software interrupt can preempt those routines when the next system tick occurs and PRD_swi is posted again.

4.1.3 A Comparison of Thread Characteristics

This table provides a quick comparison of the thread types supported by DSP/BIOS:

	HWI	**SWI**	**TSK**	**IDL**
Priority	highest	2nd highest	2nd lowest	lowest
Number of priority levels	chip-dependent	14 (plus 2 used for PRD and TSK)	15 (plus 1 used for IDL)	1
Can yield and pend	no, runs to completion except for preemption	no, runs to completion except for preemption	yes	should not; would prevent Code Composer Studio from getting target information
Execution states	inactive, ready, running	inactive, ready, running	ready, running, blocked, terminated	ready, running
Scheduler disabled by	HWI_disable	SWI_disable	TSK_disable	program exit
Posted or made ready to run by	interrupt occurs	SWI_post, SWI_andn, SWI_dec, SWI_inc, SWI_or	TSK_create	main() exits
Stack used	system stack (1 per program)	system stack (1 per program)	task stack (1 per task)	task stack used by default[†]
Context saved when preempts other thread	customizable	certain registers[‡] saved to application stack	entire context saved to task stack	--not applicable--
Context saved when blocked	--not applicable--	--not applicable--	context saved for C function calls saved to task stack	--not applicable--
Share data with thread via	streams, queues, pipes, global variables	streams, queues, pipes, global variables	streams, queues, pipes, locks, mailboxes, global variables	streams, queues, pipes, global variables

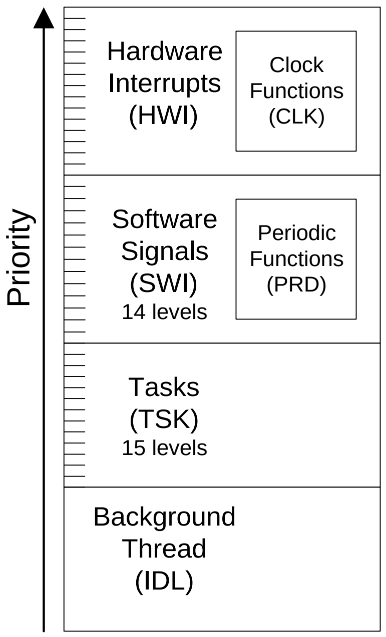
	HWI	**SWI**	**TSK**	**IDL**
Synchronize with thread via	--not applicable--	SWI mailbox	semaphores, mailboxes	--not applicable--
Function hooks	no	no	yes: create, delete, exit, task switch, ready	no
Static creation	included in default configuration template	yes	yes	yes
Dynamic creation	yes[††]	yes	yes	no
Dynamically change priority	no	yes	yes	no
Implicit logging	none	post and completion events	ready, start, block, resume, and termination events	none
Implicit statistics	monitored values	execution time	execution time	none

†. If you disable the TSK Manager in the Property dialog for the TSK Manager, IDL threads use the system stack.

‡. See section 4.3.7, *Saving Registers During Software Interrupt Preemption*, page 4-30, for a list of which registers are saved.

††. HWI objects cannot be created dynamically because they correspond to DSP interrupts. However, interrupt functions can be changed at run-time.

4.1.4 Thread Priorities



Within DSP/BIOS, hardware interrupts have the highest priority. Hardware interrupt types and priorities are determined by the chip architecture. The Configuration Tool lists HWI objects in order from highest to lowest priority. Hardware interrupts may be preempted by a higher-priority interrupt if that interrupt is enabled during the lower-priority interrupt's execution.

Software interrupts have lower priority than hardware interrupts. There are 14 priority levels available for software interrupts. Software interrupts can be preempted by a higher-priority software interrupt or any hardware interrupt. Software interrupts cannot block.

Tasks have lower priority than software interrupts. There are 15 task priority levels. Tasks can be preempted by any higher-priority thread. Tasks can also block to wait for resource availability and lower-priority threads.

The background idle loop is the thread with the lowest priority of all. It runs in a loop when the CPU is not busy running another thread.

4.1.5 Yielding and Preemption

The DSP/BIOS schedulers run the highest-priority thread that is ready to run except in the following cases:

❏ The thread that is running disables some or all hardware interrupts temporarily (with HWI_disable, the HWI dispatcher, or HWI_enter), preventing hardware ISRs from running.

❏ The thread that is running disables software interrupts temporarily (with SWI_disable). This prevents any higher-priority software interrupt from preempting the current thread. It does not prevent hardware interrupts from preempting the current thread.

❏ The thread that is running disables task scheduling temporarily (with TSK_disable). This prevents any higher-priority task from preempting the current task. It does not prevent software and hardware interrupts from preempting the current task.

- ❏ The highest-priority thread is a task that is blocked. This occurs if the task calls TSK_sleep, LCK_pend, MBX_pend, or SEM_pend.

Both hardware and software ISRs can interact with the DSP/BIOS task scheduler. When a task is blocked, it is often because the task is pending on a semaphore which is unavailable. Semaphores may be posted from ISRs as well as from other tasks. If an ISR posts a semaphore to unblock a pending task, the processor switches to that task if that task has a higher priority than the currently running task.

When running either an HWI or SWI, DSP/BIOS uses a dedicated system interrupt stack, called the *application stack*. Each task uses its own private stack. Therefore, if there are no TSK tasks in the system, all threads share the same application stack. Because DSP/BIOS uses separate stacks for each task, both the application and task stacks can be smaller. Because the application stack is smaller, you can place it in precious fast memory.

The following figure shows what happens when one type of thread is running (top row) and another thread becomes ready to run (left column). The results depend on whether or not the type of thread that is ready to run is enabled or disabled. (The action shown is that of the thread that is ready to run.)

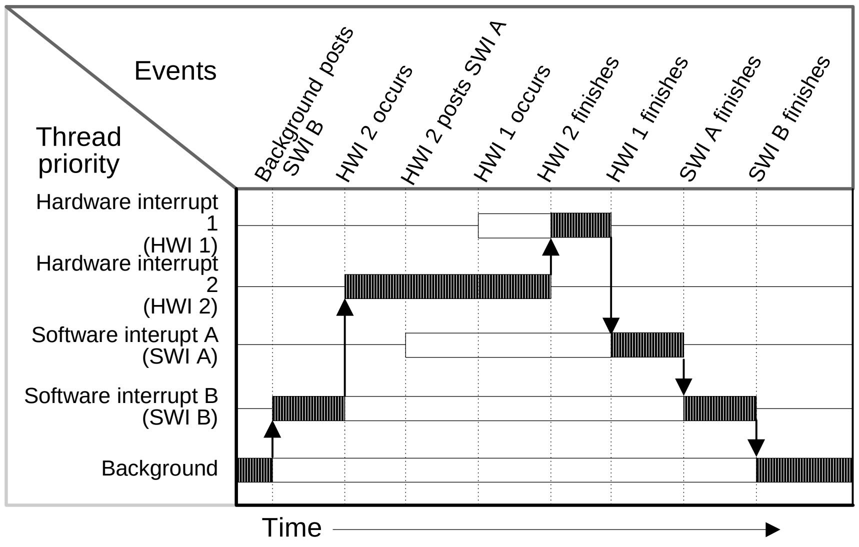
Figure 4–1 Thread Preemption

Thread Posted \ Thread Running	Hardware Interrupt (HWI)	Software Interrupt (SWI)	Task (TSK)	Idle Thread (IDL)
enabled, higher-priority HWI	**P**	**P**	**P**	**P**
disabled HWI	**W***	**W***	**W***	**W***
lower-priority HWI	**W**	**--**	**--**	**--**
enabled, higher-priority SWI	**--**	**P**	**P**	**P**
disabled SWI	**W**	**W***	**W***	**W***
same or lower-priority SWI	**W**	**W**	**--**	**--**
enabled, higher-priority TSK	**--**	**--**	**P**	**P**
disabled TSK	**W**	**W**	**W***	**W***
same or lower-priority TSK	**W**	**W**	**W**	**--**

P	= Preempts
W	= Waits
W*	= Waits until thread type or interrupt is reenabled
--	= No such object of this priority level

The following figure shows the execution graph for a scenario in which SWIs and HWIs are enabled (the default), and a hardware interrupt routine posts a software interrupt whose priority is higher than that of the software interrupt running when the interrupt occurs. Also, a higher priority hardware interrupt occurs while the first ISR is running. The second ISR is held off because the first ISR masks off (i.e., disables) the higher priority interrupt during the first ISR.

Figure 4–2 Preemption Scenario



The low priority software interrupt is asynchronously preempted by the hardware interrupts. The first ISR posts a higher-priority software interrupt, which is executed after both hardware interrupt routines finish executing.

4.2 Hardware Interrupts

Hardware interrupts handle critical processing that the application must perform in response to external asynchronous events. The DSP/BIOS HWI module is used to manage hardware interrupts.

In a typical DSP system, hardware interrupts are triggered either by on-chip peripheral devices or by devices external to the DSP. In both cases, the interrupt causes the processor to vector to the ISR address. The address to which a DSP/BIOS HWI object causes an interrupt to vector to can be either the common system HWI dispatcher or the user routine.

Hardware ISRs can be written in assembly language or a combination of both.[1] HWI functions are usually written in assembly language for efficiency. If you choose to have an HWI object use the HWI dispatcher, that object's function can also be written completely in C.

All hardware interrupts run to completion. If an HWI is posted multiple times before its ISR has a chance to run, the ISR runs only one time. For this reason, you should minimize the amount of code performed by an HWI function. If the GIE bit is enabled, a hardware interrupt may be preempted by a higher-priority interrupt that is enabled by the IEMASK.

Note that if an HWI function calls any of the PIP APIs—PIP_alloc, PIP_free, PIP_get, PIP_put—the pipe's notifyWriter or notifyReader functions run as part of the HWI context.

4.2.1 Configuring Interrupts with the Configuration Tool

In the DSP/BIOS configuration template, the HWI manager contains an HWI object for each hardware interrupt in your DSP. All HWI objects are listed in order of priority, from the highest to the lowest priority interrupt.

Using the HWI manager in the Configuration Tool, you can configure the ISR for each hardware interrupt in the DSP.

You need to enter only the name of the ISR that is called in response to a hardware interrupt in the Property Page of the corresponding HWI object in the Configuration Tool. DSP/BIOS takes care of setting up the interrupt service table so that each hardware interrupt is handled by the appropriate ISR. The Configuration Tool also allows you to select the memory segment where the interrupt service table is located.

1. C versions of hardware ISRs are followed by "()" to distinguish them from their ASM equivalents.

The on-line help in the Configuration Tool describes HWI objects and their parameters. See *HWI Module* in the *API Reference Guide* for reference information on the HWI module API calls.

4.2.2 Disabling and Enabling Hardware Interrupts

Within a software interrupt or task, you can temporarily disable hardware interrupts during a critical section of processing. The HWI_disable and HWI_enable/HWI_restore functions are used in pairs to disable and enable interrupts.

When you call HWI_disable, interrupts are globally disabled in your application. HWI_disable clears the GIE bit in the control status register (CSR), preventing the CPU from taking any maskable hardware interrupt. They therefore operate on a global basis, affecting all interrupts, as opposed to affecting individual bits in the interrupt enable register (IER).

To reenable interrupts, call HWI_enable or HWI_restore. HWI_enable always enables the GIE bit, while HWI_restore restores the value of the GIE bit to the state that existed before HWI_disable was called.

The following code examples show regions protected from all interrupts:

```
; ASM example

.include hwi.h62
...
HWI_disable B0 ; disable all interrupts, save result in reg B0
  ’do some critical operation’
HWI_restore B0


/* C example */
.include hwi.h
Uns oldmask;

oldmask = HWI_disable();
  ’do some critical operation; ’
  ’do not call TSK_sleep(), SEM_post, etc.’
HWI_restore(oldmask);
```

Using HWI_restore instead of HWI_enable allows the pair of calls to be nested. If the calls are nested, the outermost call to HWI_disable turns interrupts off, and the innermost call to HWI_disable does nothing. Interrupts are not reenabled until the outermost call to HWI_restore. Be careful when using HWI_enable because this call enables interrupts even if they were already disabled when HWI_disable was called.

Note:

DSP/BIOS kernel calls that may cause task rescheduling (for example, SEM_post() and TSK_sleep()) should be avoided within a block surrounded by HWI_disable() and HWI_enable() since the interrupts may be disabled for an indeterminate amount of time if a task switch occurs.

4.2.3 Context and Interrupt Management within Interrupts

When a hardware interrupt preempts the function that is currently executing, the HWI function must save and restore any registers it uses or modifies. DSP/BIOS provides the HWI_enter assembly macro to save registers and the HWI_exit assembly macro to restore registers. Using these macros gives the function that was preempted the same context when it resumes running. These macros also handle which interrupts are disabled while the ISR runs.

In order to support interrupt routines written completely in C, DSP/BIOS provides an HWI dispatcher that performs these enter and exit macros for an interrupt routine. An ISR may handle context saving and interrupt disabling using this HWI dispatcher or by explicitly calling HWI_enter and HWI_exit. The Configuration Tool allows you to choose whether the HWI dispatcher is used for individual HWI objects.

The HWI dispatcher is the preferred method for handling interrupts. If an HWI object does not use the dispatcher, the HWI_enter assembly macro must be called prior to any DSP/BIOS API calls that could post or affect a software interrupt or semaphore. The HWI_exit assembly macro must be called at the very end of the function’s code.

The system HWI dispatcher, in effect, calls the configured ISR function from within an HWI_enter/HWI_exit macro pair. This allows the HWI function to be written completely in C. It would, in fact, cause a system crash were the dispatcher to call a function that contains the HWI_enter/HWI_exit macro pair. Using the dispatcher therefore allows for only one instance of the HWI_enter and HWI_exit code.

Whether called explicitly or by the HWI dispatcher, the HWI_enter and HWI_exit macros prepare an ISR to call any C function. In particular, the ISR is prepared to call any DSP/BIOS API function that is allowed to be called from the context of an HWI. (See section A.1, *Functions Callable by Tasks, SWI Handlers, or Hardware ISRs* in the *API Reference Guide* for a complete list of these functions.)

Note:

When using the system HWI dispatcher, the ISR code must not call HWI_enter and HWI_exit.

Regardless of which HWI dispatching method is used, DSP/BIOS uses the application stack during the execution of both SWIs and HWIs. If there are no TSK tasks in the system, this application stack is used by all threads. If there are TSK tasks, each task uses its own private stack. Whenever a task is preempted by an SWI or HWI, DSP/BIOS uses the application stack for the duration of the interrupt thread.

HWI_enter and HWI_exit both take four parameters:

- The first two, ABMASK and CMASK, specify which A, B, and control registers are to be saved and restored by the ISR.
- The third parameter, IEMASK, is a mask of those interrupts that are to be disabled between the HWI_enter and HWI_exit macro calls.

  When an interrupt is triggered, the processor disables interrupts globally (by clearing the GIE bit in the control status register (CSR)) and then jumps to the ISR set up in the interrupt service table. The HWI_enter macro reenables interrupts by setting the GIE in the CSR. Before doing so, HWI_enter selectively disables bits in the interrupt enable register (IER) determined by the IEMASK parameter. Hence, HWI_enter gives you control to select what interrupts can and cannot preempt the current HWI function.

  When HWI_exit is called, the bit pattern in the IEMASK determines what interrupts are restored by HWI_exit by setting the corresponding bits in the IER. Of the interrupts in IEMASK, HWI_exit restores only those that were disabled with HWI_enter. If upon exiting the ISR you do not wish to restore one of the interrupts that was disabled with HWI_enter, do not set that interrupt bit in IEMASK in HWI_exit. HWI_exit does not affect the status of interrupt bits that are not in IEMASK.
- The fourth parameter, CCMASK, specifies the value to place in the cache control field of the CSR. This cache state remains in effect for the duration of code executed between the HWI_enter and HWI_exit calls. Some typical values for this mask are defined in c62.h62 (e.g., C62_PCC_ENABLE). You can OR the PCC code and DCC code together to generate CCMASK. If you use 0 as CCMASK, a default value is used. You set this value in the Global Settings Properties in the Configuration Tool by right-clicking and selecting Properties.

CLK_F_isr, which handles one of the on-chip timer interrupts when the Clock Manager is enabled, also uses the default cache value set by the Configuration Tool. HWI_enter saves the current CSR status before it sets the cache bits as defined by CCMASK. HWI_exit restores CSR to its value at the interrupted context.

The predefined masks C62_ABTEMPS and C62_CTEMPS specify all of the C language temporary A/B registers and all of the temporary control registers, respectively. These masks may be used to save the registers that may be freely used by a C function. When using the HWI dispatcher, there is no ability to specify a register set, so the registers specified by these masks are all saved and restored.

For example, if your ISR calls a C function you would use:

```
HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
 'isr code'
HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
```

HWI_enter (defined in hwi.h62) should be used to save all of the C runtime environment registers before calling any C or DSP/BIOS functions. HWI_exit should be used to restore these registers.

In addition to saving and restoring the C runtime environment registers, HWI_enter and HWI_exit make sure the DSP/BIOS scheduler is called only by the outermost interrupt routine if nested interrupts occur. If the ISR or another nested ISR triggers an SWI handler with SWI_post(), or readies a higher priority task (e.g., by calling SEM_ipost() or TSK_itick()), the outermost HWI_exit invokes the SWI and TSK schedulers. The SWI scheduler services all pending SWI handlers before performing a context switch to a higher priority task (if necessary).

See section A.1, *Functions Callable by Tasks, SWI Handlers, or Hardware ISRs* in the *API Reference Guide* for a complete list of functions that may be called by an ISR.

Note:

HWI_enter and HWI_exit must surround all statements in any DSP/BIOS assembly or C language ISRs that reference DSP/BIOS functions. Using the HWI dispatcher satisfies this requirement.

```
;
; ======== myclk.s62 ========
;

    .include "hwi.h62"  ; macro header file

IEMASK    .set 0
CCMASK    .set c62_PCC_DISABLE
    .text

;
; ======== myclkisr ========
;
    global _myclkisr
_myclkisr:

    ; save all C runtime environment registers
    HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK

    b       _TSK_itick   ; call TSK itick (C function)
    mvkl    tiret, b3
    mvkh    tiret, b3

    nop     3

tiret:

    ; restore saved registers and call DSP/BIOS scheduler
    HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK

    .end
```

4.2.4 Registers

This section summarizes information found in the *TMS320C6000 Optimizing C Compiler User's Guide*.

Role	Register(s)	Explanation
Stack Pointer	B15 (SP)	Points to the top location of the downward-growing stack.
Frame Pointer	A15 (FP)	Points to the bottom of the frame for the current function.
Data page pointer	B14 (DP)	Points to the beginning of the `.bss` section. Small memory model data and 'near' data are addressed by offsets relative to the DP.
Structure pointer	A3	Pointer to a returned structure.
Return Value	A4, (A5)	For integer, pointer, (double or long) values.
Return Address	B3	Function return address.
Register Parameters	A4(A5), B4(B5), A6(A7), B6(B7), A8(A9), B8(B9), A10(A11), B10(B11), A12(A13), B12(B13)	Function parameters are passed in these registers. See the compiler manual for details.
Control Register File	AMR CSR ICR IER IFR IN ISR OUT	Addressing Mode Register Control Status Register Interrupt Clear Register Interrupt Enable Register Interrupt Flag Register Input Register Interrupt Set Register Output Register

The table below shows which registers are saved and restored by functions conforming to the Texas Instruments TMS320C6000 C runtime model.

A0	A1	A2	A3	A4	A5	A6	A7
A8	A9	▨ A10	▨ A11	▨ A12	▨ A13	▨ A14	▨ A15
B0	B1	B2	B3	B4	B5	B6	B7
B8	B9	▨ B10	▨ B11	▨ B12	▨ B13	▨ B14	▨ B15

AMR	■ CSR	■ ICR	■ IER	■ IFR	■ IN	■ ISR	■ OUT

Key:

□	Scratch Registers; caller presumes they are altered
▨	Saved and restored by a C callee
■	Control Registers global to all tasks

4.3 Software Interrupts

Software interrupts are patterned after hardware ISRs. The SWI module in DSP/BIOS provides a software interrupt capability. Software interrupts are triggered programmatically, through a call to a DSP/BIOS API such as SWI_post. Software interrupts have priorities that are higher than tasks but lower than hardware interrupts.

The SWI module should not be confused with the “SWI” instruction that exists on many processors. The DSP/BIOS SWI module is independent from any processor-specific software interrupt features.

SWI threads are suitable for handling application tasks that occur at slower rates or are subject to less severe real-time deadlines than those of hardware interrupts.

The DSP/BIOS APIs that can trigger or post a software interrupt are:

❏ SWI_andn
❏ SWI_dec
❏ SWI_inc
❏ SWI_or
❏ SWI_post

The SWI manager controls the execution of all software interrupts. When the application calls one of the APIs above, the SWI manager schedules the function corresponding to the software interrupt for execution. To handle all software interrupts in an application, the SWI manager uses SWI objects.

If a software interrupt is posted, it runs only after all pending hardware interrupts have run. An SWI routine in progress may be preempted at any time by a hardware ISR; the hardware ISR completes before the SWI handler resumes. On the other hand, SWI handlers always preempt tasks. All pending software interrupts run before even the highest priority task is allowed to run. In effect, an SWI handler is like a task with a priority higher than all ordinary tasks.

Note:

An SWI handler runs to completion unless it is interrupted by a hardware interrupt or preempted by a higher priority SWI.

4.3.1 Creating SWI Objects

As with many other DSP/BIOS objects, you can create SWI objects either dynamically (with a call to SWI_create()) or statically (with the Configuration Tool). Software interrupts you create dynamically can also be deleted during program execution.

To add a new software interrupt with the Configuration Tool, create a new SWI object for the SWI manager in the Configuration Tool. In the Property Page of each SWI object, you can set the function each software interrupt is to run when the object is triggered by the application. The Configuration Tool also allows you to enter two arguments for each SWI function.

In the Property Page of the SWI manager, you can determine from which memory segment SWI objects are allocated. SWI objects are accessed by the SWI manager when software interrupts are posted and scheduled for execution.

The on-line help in the Configuration Tool describes SWI objects and their parameters. See *SWI Module* in the *API Reference Guide* for reference information on the SWI module API calls.

To create a software interrupt dynamically, use a call like the following:

```
swi = SWI_create(attrs);
```

Here, swi is the interrupt handle and the variable attrs points to the SWI attributes. The SWI attribute structure (of type SWI_Attrs) contains all those elements that can be configured for an SWI using the Configuration Tool. attrs can be NULL, in which case, a default set of attributes is used. Typically, attrs contains at least a function for the handler.

Note:

SWI_create() can only be called from the task level, not from an HWI or another SWI.

SWI_getattrs can be used to retrieve all the SWI_Attrs attributes. Some of these attributes can change during program execution, but typically they contain the values assigned when the object was created.

```
SWI_getattrs(swi, attrs);
```

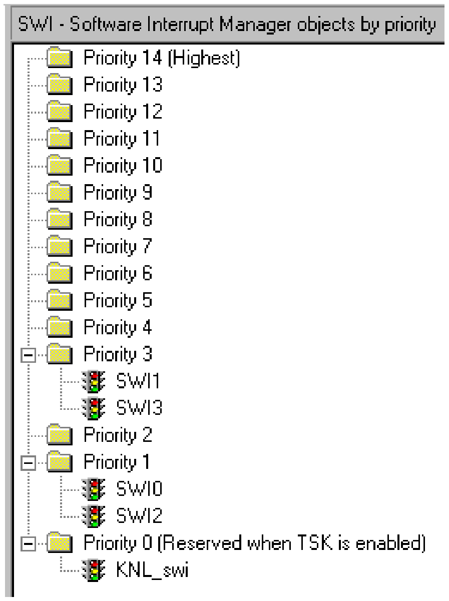
4.3.2 Setting Software Interrupt Priorities in the Configuration Tool

There are different priority levels among software interrupts. You can create as many software interrupts as your memory constraints allow for each priority level. You can choose a higher priority for a software interrupt that handles a thread with a shorter real-time deadline, and a lower priority for a software interrupt that handles a thread with a less critical execution deadline.

To set software interrupt priorities with the Configuration Tool, follow these steps:

1) In the Configuration Tool, highlight the Software Interrupt Manager. Notice the SWI objects in the right half of the window organized by priority in priority level folders. (If you do not see a list of SWI objects in the right half of the window, right-click on the SWI manager, then choose View→Ordered collection view.)

2) To change the priority of a SWI object, drag the software interrupt to the folder of the corresponding priority. For example, to change the priority of SWI0 to 3, select it with the mouse and drag it to the folder labeled "Priority 3"

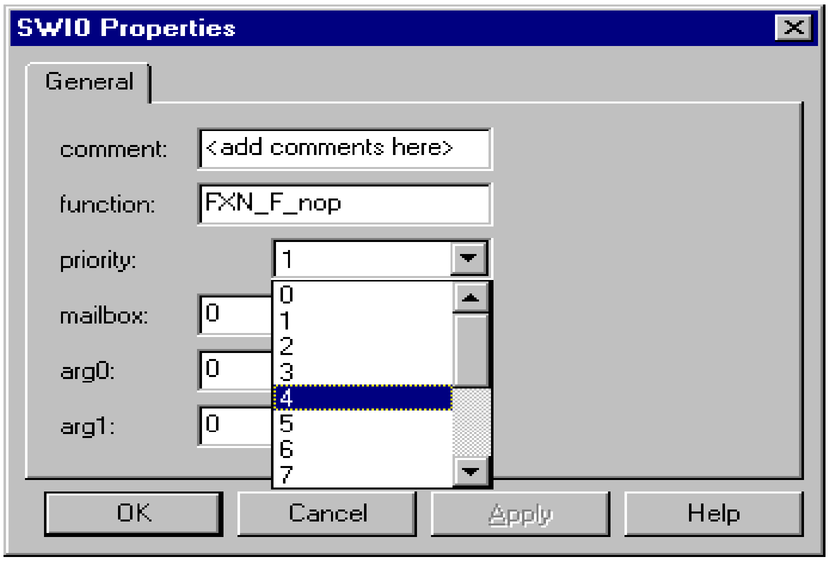


Notes:

❏ Software interrupts can have up to 15 priority levels. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler. See section 4.3.3, *Software Interrupt Priorities and Application Stack Size*, page 4-21, for stack size restrictions.

❏ You cannot sort software interrupts within a single priority level.

❑ The Property window for an SWI object shows its numeric priority level (from 0 to 14; 14 is the highest level). You can also set the priority by selecting the priority level from the menu in the Property window.


SWI0 Properties

comment: <add comments here>

function: FXN_F_nop

priority: 1

mailbox: 0

arg0: 0

arg1: 0

4.3.3 Software Interrupt Priorities and Application Stack Size

All threads in DSP/BIOS, excluding tasks, are executed using the same software stack (the application stack).

The application stack stores the register context when a software interrupt preempts another thread. To allow the maximum number of preemptions that may occur at run time, the required stack size grows each time you add a software interrupt priority level. Thus, giving software interrupts the same priority level is more efficient in terms of stack size than giving each software interrupt a separate priority.

The default application stack size for the MEM module is 256 words. You can change the sizes in the Configuration Tool. The estimated sizes required are shown in the status bar at the bottom of the Configuration Tool.

You can have up to 15 software interrupt priority levels, but each level requires a larger application stack. If you see a pop-up message that says “the application stack size is too small to support a new software interrupt priority level,” increase the Application Stack Size property of the Memory Section Manager.

Creating the first PRD object creates a new SWI object called PRD_swi (see section 4.9, *Periodic Function Manager (PRD) and the System Clock*, page

4-63, for more information on PRD). If no SWI objects have been created before the first PRD object is added, adding PRD_swi uses the first priority level, producing a corresponding increase in the required application stack.

If the TSK manager has been enabled, the TSK scheduler (run by an SWI object named KNL_swi) reserves the lowest SWI priority level. No other SWI objects can have that priority.

4.3.4 Execution of Software Interrupts

Software interrupts can be scheduled for execution with a call to SWI_andn, SWI_dec, SWI_inc, SWI_or, and SWI_post. These calls can be used virtually anywhere in the program—interrupt service routines, periodic functions, idle functions, or other software interrupt functions.

When an SWI object is posted, the SWI manager adds it to a list of posted software interrupts that are pending execution. Then the SWI manager checks whether software interrupts are currently enabled. If they are not, as is the case inside an HWI function, the SWI manager returns control to the current thread.

If software interrupts are enabled, the SWI manager checks the priority of the posted SWI object against the priority of the thread that is currently running. If the thread currently running is the background idle loop or a lower priority SWI, the SWI manager removes the SWI from the list of posted SWI objects and switches the CPU control from the current thread to start execution of the posted SWI function.

If the thread currently running is an SWI of the same or higher priority, the SWI manager returns control to the current thread, and the posted SWI function runs after all other SWIs of higher priority or the same priority that were previously posted finish execution.

Note:

When an SWI starts executing it must run to completion without blocking.

SWI functions can be preempted by threads of higher priority (such as an HWI or an SWI of higher priority). However, SWI functions cannot block. You cannot suspend a software interrupt while it waits for something—like a device—to be ready.

If an SWI is posted multiple times before the SWI manager has removed it from the posted SWI list, its SWI function executes only once, much like an ISR is executed only once if the hardware interrupt is triggered multiple times before the CPU clears the corresponding interrupt flag bit in the interrupt flag register. (See section 4.3.5, *Using an SWI Object's Mailbox*, page 4-23, for more information on how to handle SWIs that are posted multiple times before they are scheduled for execution.)

Applications should not make any assumptions about the order in which SWI handlers of equal priority are called. However, an SWI handler may safely post itself (or be posted by another interrupt). If more than one is pending, all SWI handlers are called before any tasks run.

4.3.5 Using an SWI Object's Mailbox

Each SWI object has a 32-bit mailbox, which is used either to determine whether to post the software interrupt or as a value that can be evaluated within the SWI function.

SWI_post, SWI_or, and SWI_inc post an SWI object unconditionally:

❏ SWI_post does not modify the value of the SWI object mailbox when it is used to post a software interrupt.

❏ SWI_or sets the bits in the mailbox determined by a mask that is passed as a parameter, and then posts the software interrupt.

❏ SWI_inc increases the SWI's mailbox value by one before posting the SWI object.

SWI_andn and SWI_dec post the SWI object only if the value of its mailbox becomes 0:

❏ SWI_andn clears the bits in the mailbox determined by a mask passed as a parameter.

❏ SWI_dec decreases the value of the mailbox by one.

The following table summarizes the differences between these functions:

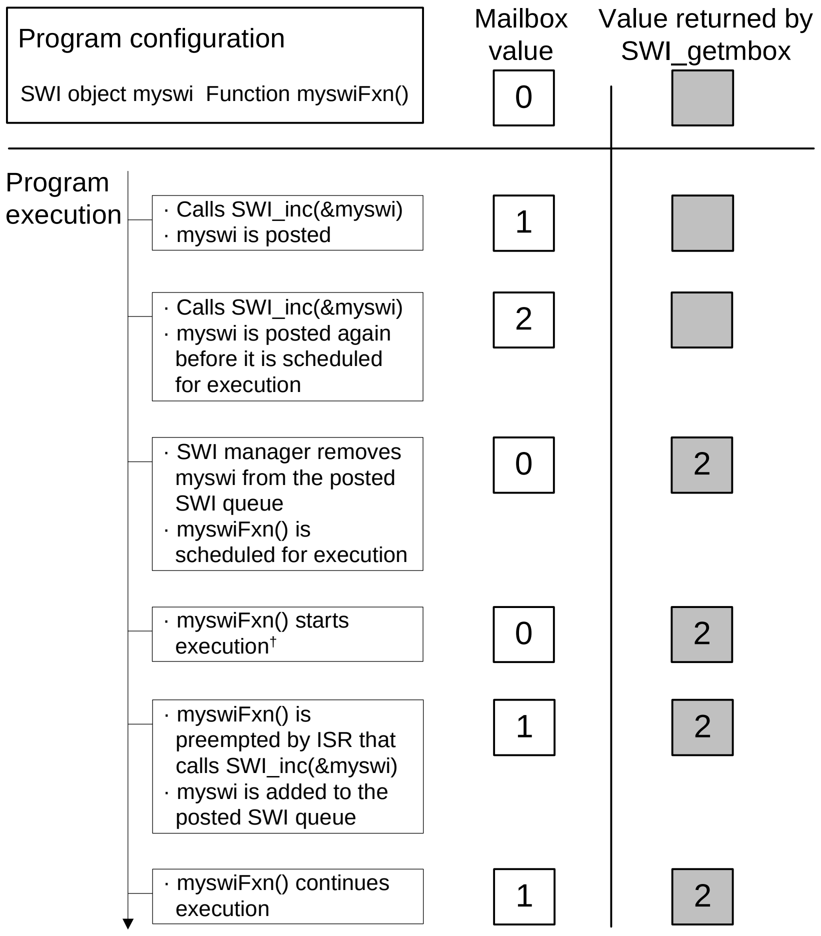
	Treat mailbox as bitmask	Treat mailbox as counter	Does not modify mailbox
Always post	SWI_or	SWI_inc	SWI_post
Post if becomes 0	SWI_andn	SWI_dec	

The SWI mailbox allows you to have a tighter control over the conditions that should cause an SWI function to be posted or the number of times the SWI function should be executed once the software interrupt is posted and scheduled for execution.

To access the value of its mailbox, an SWI function can call SWI_getmbox. SWI_getmbox can be called only from the SWI's object function. The value returned by SWI_getmbox is the value of the mailbox before the SWI object was removed from the posted SWI queue and the SWI function was scheduled for execution. When the SWI manager removes a pending SWI object from the posted object's queue, its mailbox is reset to its initial value. The initial value of the mailbox is set from the Property Page when the SWI object is created with the Configuration Tool. If while the SWI function is executing it is posted again, its mailbox is updated accordingly. However, this does not affect the value returned by SWI_getmbox while the SWI functions execute. That is, the mailbox value that SWI_getmbox returns is the latched mailbox value when the software interrupt was removed from the list of pending SWIs. The SWI's mailbox however, is immediately reset after the SWI is removed from the list of pending SWIs and scheduled for execution. This gives the application the ability to keep updating the value of the SWI mailbox if a new posting occurs, even if the SWI function has not finished its execution.

For example, if an SWI object is posted multiple times before it is removed from the queue of posted SWIs, the SWI manager schedules its function to execute only once. However, if an SWI function must always run multiple times when the SWI object is posted multiple times, SWI_inc should be used to post the SWI. When an SWI has been posted using SWI_inc, once the SWI manager calls the corresponding SWI function for execution, the SWI function can access the SWI object mailbox to know how many times it was posted before it was scheduled to run, and proceed to execute the same routine as many times as the value of the mailbox.

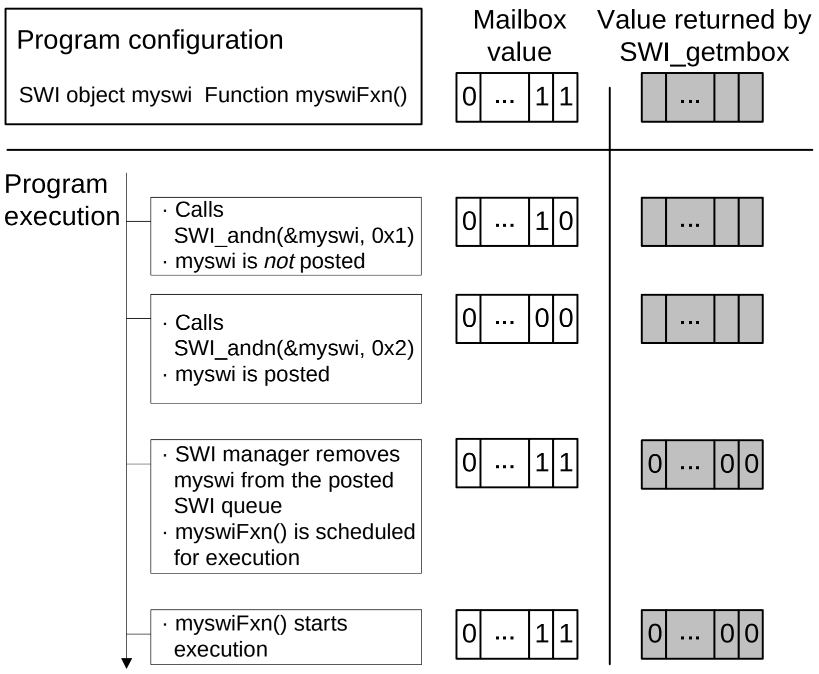
Figure 4–3 Using SWI_inc to Post an SWI


Program configuration
SWI object myswi Function myswiFxn()
Mailbox value
Value returned by SWI_getmbox
0
Program execution
· Calls SWI_inc(&myswi)
· myswi is posted
1
· Calls SWI_inc(&myswi)
· myswi is posted again before it is scheduled for execution
2
· SWI manager removes myswi from the posted SWI queue
· myswiFxn() is scheduled for execution
0
2
· myswiFxn() starts execution†
0
2
· myswiFxn() is preempted by ISR that calls SWI_inc(&myswi)
· myswi is added to the posted SWI queue
1
2
· myswiFxn() continues execution
1
2

```
† myswiFxn()
 {   . . .
     repetitions = SWI_getmbox();
     while (repetitions --){
         ‘run SWI routine‘
     }
 . . .
 }
```

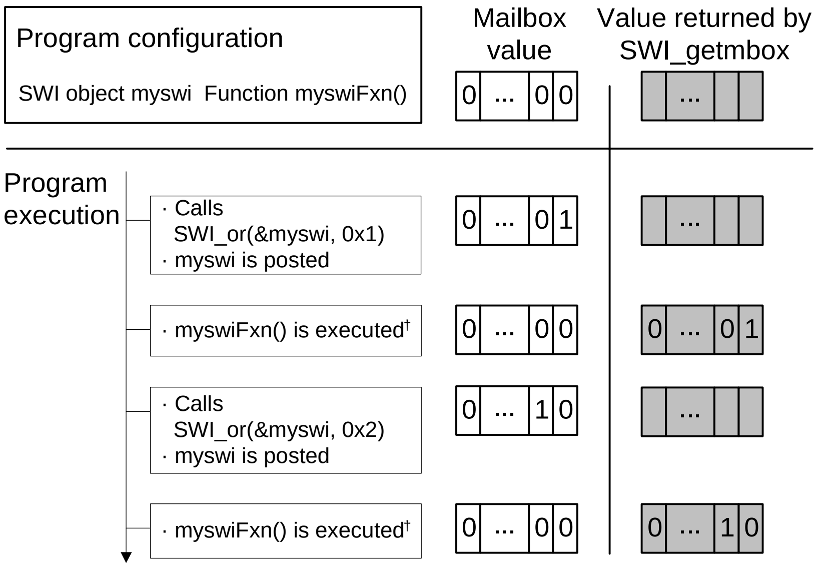
If more than one event must always happen for a given software interrupt to be triggered, SWI_andn should be used to post the corresponding SWI object. For example, if a software interrupt must wait for input data from two different devices before it can proceed, its mailbox should have two set bits when the SWI object was created with the Configuration Tool. When both routines that provide input data have completed their tasks, they should both call SWI_andn with complementary bitmasks that clear each of the bits set in the SWI mailbox default value. Hence, the software interrupt is posted only when data from both processes is ready.

Figure 4–4 Using SWI_andn to Post an SWI


Program configuration
SWI object myswi Function myswiFxn()
Mailbox value
Value returned by SWI_getmbox
0 ... 1 1
...
Program execution
· Calls SWI_andn(&myswi, 0x1)
· myswi is *not* posted
0 ... 1 0
· Calls SWI_andn(&myswi, 0x2)
· myswi is posted
0 ... 0 0
· SWI manager removes myswi from the posted SWI queue
· myswiFxn() is scheduled for execution
0 ... 1 1
0 ... 0 0
· myswiFxn() starts execution
0 ... 1 1
0 ... 0 0

In some situations the SWI function may call different routines depending on the event that posted it. In that case the program can use SWI_or to post the SWI object unconditionally when an event happens. The value of the bitmask used by SWI_or encodes the event type that triggered the post operation, and can be used by the SWI function as a flag that identifies the event and serves to choose the routine to execute

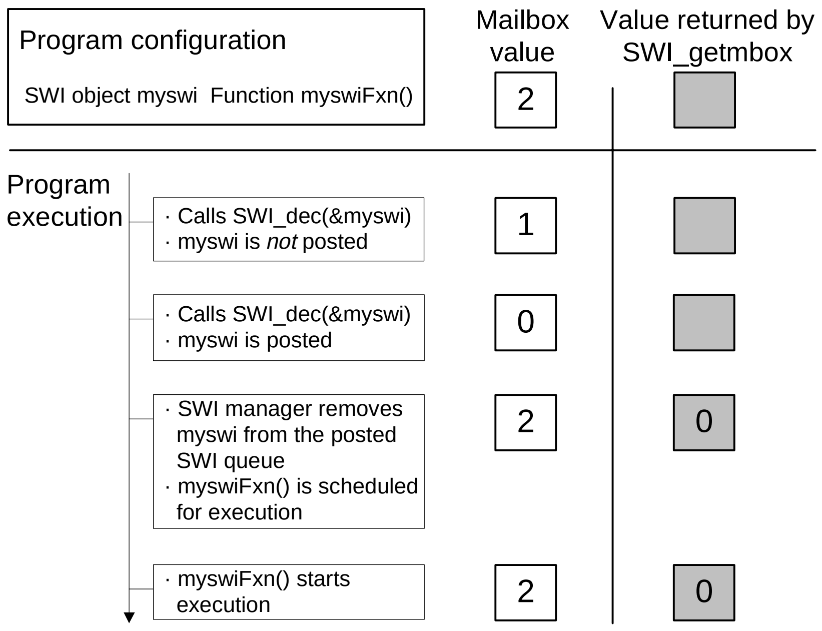
Figure 4–5 Using SWI_or to Post an SWI.


Program configuration
SWI object myswi Function myswiFxn()
Mailbox value
Value returned by SWI_getmbox
0 ... 0 0
...
Program execution
· Calls SWI_or(&myswi, 0x1)
· myswi is posted
0 ... 0 1
· myswiFxn() is executed†
0 ... 0 0
0 ... 0 1
· Calls SWI_or(&myswi, 0x2)
· myswi is posted
0 ... 1 0
· myswiFxn() is executed†
0 ... 0 0
0 ... 1 0

```
† myswiFxn()
 {
     ...
     eventType = SWI_getmbox();
     switch (eventType) {
         case '0x1':
             'run processing algorithm 1'
         case '0x2':
             'run processing algorithm 2'
         case '0x4':
             'run processing algorithm 3'
         ...
     }
     ...
 }
```

If the program execution requires that multiple occurrences of the same event must take place before an SWI is posted, SWI_dec should be used to post the SWI. By configuring the SWI mailbox to be equal to the number of occurrences of the event before the SWI should be posted and calling SWI_dec every time the event occurs, the SWI is posted only after its mailbox reaches 0; i.e., after the event has occurred a number of times equal to the mailbox value.

Figure 4–6 Using SWI_dec to Post an SWI


Program configuration
SWI object myswi Function myswiFxn()
Mailbox value
Value returned by SWI_getmbox
2
Program execution
· Calls SWI_dec(&myswi)
· myswi is *not* posted
1
· Calls SWI_dec(&myswi)
· myswi is posted
0
· SWI manager removes myswi from the posted SWI queue
· myswiFxn() is scheduled for execution
2
0
· myswiFxn() starts execution
2
0

4.3.6 Benefits and Tradeoffs

There are two main benefits to using software interrupts instead of hardware interrupts.

First, SWI handlers can execute with all hardware interrupts enabled. To understand this advantage, recall that a typical hardware ISR modifies a data structure that is also accessed by tasks. Tasks therefore need to disable hardware interrupts when they wish to access these data structures in a mutually exclusive way. Obviously, disabling hardware interrupts always has the potential to degrade the performance of a real-time system.

Conversely, if a shared data structure is modified by an SWI handler instead of a hardware ISR, mutual exclusion can be achieved by disabling software interrupts while the task accesses the shared data structure (SWI_disable() and SWI_enable() are described later in this chapter). Thus, there is no effect on the ability of the system to respond to events in real-time using hardware interrupts.

It often makes sense to break “long” ISRs into two pieces. The hardware ISR takes care of the extremely time-critical operation and defers the less critical processing to an SWI handler.

The second advantage is that an SWI handler can call some functions that cannot be called from a hardware ISR, because an SWI handler is guaranteed not to run while DSP/BIOS is updating internal data structures. This is an important feature of DSP/BIOS and you should become familiar with the table in section A.1, *Functions Callable by Tasks, SWI Handlers, or Hardware ISRs* in the *API Reference Guide* that lists DSP/BIOS functions and the threads from which each function may be called.

Note:

SWI handlers can call any DSP/BIOS function that does not block. For example, SEM_pend() can make a task block, so SWI handlers cannot call SEM_pend() or any function that calls SEM_pend() (e.g., MEM_alloc(), TSK_sleep()).

For example, the function TSK_setpri() changes the priority of a task after it has been created. TSK_setpri() cannot be called from a hardware ISR, but it can be called from a task or an SWI handler.

To understand why this is important, consider an example where the priority of one or more tasks needs to be altered depending on the system’s response to some external event as signaled by a hardware interrupt. Since TSK_setpri() cannot be called from the hardware ISR, the system would either need to: 1) ready a task where TSK_setpri() could be called, or 2) the hardware ISR could use SWI_post() to trigger an SWI handler. The SWI handler, in turn, can call TSK_setpri().

There are two major advantages to using an SWI handler instead of a TSK task in the above situation. The first advantage is that an extra task, whose only purpose is to call TSK_setpri() on behalf of the ISR, is not required. Since each task requires its own stack, less memory is required.

The second advantage is that SWI handlers require less C context to be saved. Therefore, SWI handlers can also be switched more efficiently than tasks.

On the other hand, an SWI handler must complete before any blocked task is allowed to run. There might be situations where the use of a task might fit better with the overall system design, in spite of any additional overhead involved.

4.3.7 Saving Registers During Software Interrupt Preemption

When a software interrupt preempts another thread, DSP/BIOS preserves the context of the preempted thread by automatically saving all of the following CPU registers onto the application stack:

a0	a5	b0	b5	
a1	a6	b1	b6	CSR
a2	a7	b2	b7	AMR
a3	a8	b3	b8	
a4	a9	b4	b9	

Your SWI function does not need to save and restore any of these registers, even if the SWI function is written in assembly.

However, SWI functions that are written in assembly must follow C register usage conventions: they must save and restore any of the registers numbered A10 to A15 and B10 to B15. (See the *TMS320C6000 Optimizing C Compiler User's Guide* for more details on C register conventions.) The data page register (B14) is initialized at program startup to the start address of .bss and must not be modified.

An SWI function that modifies the IER register should save it and then restore it before it returns. If the SWI function fails to do this, the change becomes permanent and any other thread that starts to run or that the program returns to afterwards can inherit the modification to the IER.

The context is not saved automatically within an HWI function. You must use either the HWI dispatcher or the HWI_enter and HWI_exit macros to preserve the interrupted context when an HWI function is triggered.

4.3.8 Synchronizing SWI Handlers

Within an idle loop function, task, or software interrupt function, you can temporarily prevent preemption by a higher-priority software interrupt by calling SWI_disable(), which disables all SWI preemption. To reenable SWI preemption, call SWI_enable().

Software interrupts are enabled or disabled as a group. An individual software interrupt cannot be enabled or disabled on its own.

When DSP/BIOS finishes initialization and before the first task is called, software interrupts have been enabled. If an application wishes to disable software interrupts, it calls SWI_disable() as follows:

```
key = SWI_disable();
```

The corresponding enable function is SWI_enable().

```
SWI_enable(key);
```

key is a value used by the SWI module to determine if SWI_disable() has been called more than once. This allows nesting of SWI_disable() / SWI_enable() calls, since only the outermost SWI_enable() call actually enables software interrupts. In other words, a task can disable and enable software interrupts without having to determine if SWI_disable() has already been called elsewhere.

When software interrupts are disabled, a posted software interrupt does not run at that time. Instead, the interrupt is “latched” in software and runs when software interrupts are enabled and it is the highest-priority thread that is ready to run.

Note:

An important side effect of SWI_disable() is that task preemption is also disabled. This is because DSP/BIOS uses software interrupts internally to manage semaphores and clock ticks.

To delete a dynamically created software interrupt, use SWI_delete().

```
SWI_delete(swi);
```

The memory associated with swi is freed. SWI_delete() can only be called from the task level.

4.3.9 Example—SWI Time-Sharing

In this section, we show how to implement a simple time-sharing scheme using software interrupts and a dedicated clock. The example program switest.c is included on the DSP/BIOS product disk. The method shown below is designed primarily to illustrate some features of software interrupts and should not be taken as a fully worked-out time-sharing system.

In this example, 3 tasks have been created with the Configuration Tool. Each task has a computation loop consisting of a LOG_printf() that prints the name of the task, and then pends on one of three semaphores. A CLK object and an SWI object have also been created with the Configuration Tool. The function for the CLK object posts the SWI that in turn checks the time marked by the system clock and posts some of the semaphores depending on the current system time. This effectively causes a time sharing scheme among these tasks of equal priority.

Note that clkFxn() runs as part of a hardware ISR. Instead of doing all the processing required to determine what task to wake up, clkFxn() posts the SWI object. This in turn causes SwiFxn() to run, and the processing takes place in the context of a SWI object, which can be preempted by other hardware interrupts.

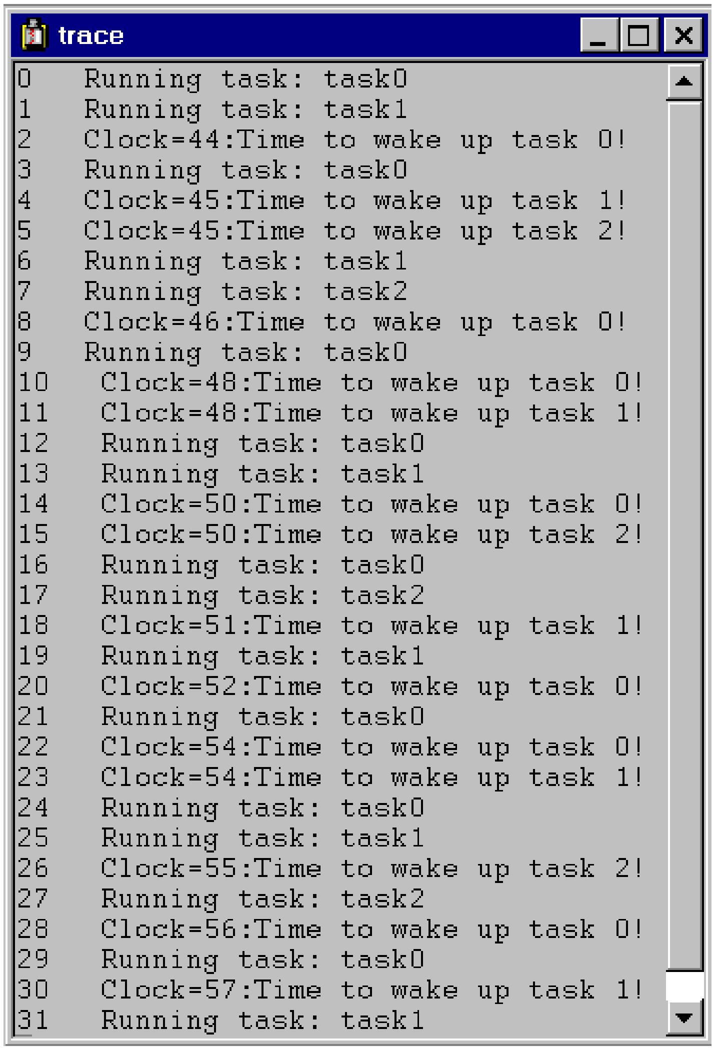
The program cycles through the three tasks. The length of time between each task switch depends on the period of the dedicated clock. If the period is increased, the time between task switches increases. In other words, each task runs for a longer slice of time.

```
/*
 * ======== switest.c =======
 * In this example, 3 tasks have been created with the
 * Configuration Tool. Each task has a computation loop
 * consisting of a LOG_printf() that prints the name of the
 * task, and then pends on one of 3 semaphores. A CLK object
 * and SWI object have also been created with the Config Tool.
 * The function for the CLK object posts the SWI that in turn
 * checks the time marked by the system clock and post some
 * of the semaphores depending on the current system time. This
 * effectively causes a time sharing scheme among these tasks
 * of equal priority.
 */

#include <std.h>
#include <clk.h>
#include <log.h>
#include <sem.h>
#include <swi.h>
#include <sys.h>
#include <tsk.h>

#define SWITCH0 2
#define SWITCH1 3
#define SWITCH2 5

extern LOG_Obj trace;
extern SEM_Obj sem0;
extern SEM_Obj sem1;
extern SEM_Obj sem2;
extern SWI_Obj swiSlice;

Void clkFxn(Void);
Void swiFxn(Void);
Void taskFxn0(Void);
Void taskFxn1(Void);
Void taskFxn2(Void);
```

```
/*
 *  ======== main ========
 */
Void main(Int argc, Char *argv[])
{
    LOG_printf(&trace,"Switest example started!\n");
}

/*  ======== clkFxn ========
 *  This function is called from the timer ISR. */
Void clkFxn(Void)
{
    SWI_post(&swiSlice);
}

/*  ======== swiFxn ======== */
Void swiFxn(Void)
{
    if ((CLK_getltime()%SWITCH0) == 0) {
       LOG_printf(&trace,"Clock=%d:Time to wake up task 0!",
CLK_getltime());
        SEM_post(&sem0);
    }
    if ((CLK_getltime()%SWITCH1) == 0) {
       LOG_printf(&trace,"Clock=%d:Time to wake up task 1!",
CLK_getltime());
        SEM_post(&sem1);
    }
    if ((CLK_getltime()%SWITCH2) == 0) {
       LOG_printf(&trace,"Clock=%d:Time to wake up task 2!",
CLK_getltime());
        SEM_post(&sem2);
    }
}

/*  ======== taskFxn0 ======== */
Void taskFxn0(Void)
TSK_settime(TSK_self());
{
    for (;;) {
        LOG_printf(&trace,"Running task: %s",
TSK_getname(TSK_self());
        TSK_deltatime(TSK_self());
        SEM_pend(&sem0,SYS_FOREVER);
    }
}
```

```
/*  ======== taskFxn1 ======== */
Void taskFxn1(Void)
TSK_settime(TSK_self());
{
    for (;;) {
        LOG_printf(&trace,"Running task: %s",
TSK_getname(TSK_self()));
        TSK_deltatime(TSK_self());
        SEM_pend(&sem1,SYS_FOREVER);
    }
}

/*  ======== taskFxn2 ======== */
Void taskFxn2(Void)
TSK_settime(TSK_self());
{
    for (;;) {
        LOG_printf(&trace,"Running task: %s",
TSK_getname(TSK_self()));
        TSK_deltatime(TSK_self());
        SEM_pend(&sem2,SYS_FOREVER);
    }
}
```

The output from the test should resemble the following:



```
0    Running task: task0
1    Running task: task1
2    Clock=44:Time to wake up task 0!
3    Running task: task0
4    Clock=45:Time to wake up task 1!
5    Clock=45:Time to wake up task 2!
6    Running task: task1
7    Running task: task2
8    Clock=46:Time to wake up task 0!
9    Running task: task0
10    Clock=48:Time to wake up task 0!
11    Clock=48:Time to wake up task 1!
12    Running task: task0
13    Running task: task1
14    Clock=50:Time to wake up task 0!
15    Clock=50:Time to wake up task 2!
16    Running task: task0
17    Running task: task2
18    Clock=51:Time to wake up task 1!
19    Running task: task1
20    Clock=52:Time to wake up task 0!
21    Running task: task0
22    Clock=54:Time to wake up task 0!
23    Clock=54:Time to wake up task 1!
24    Running task: task0
25    Running task: task1
26    Clock=55:Time to wake up task 2!
27    Running task: task2
28    Clock=56:Time to wake up task 0!
29    Running task: task0
30    Clock=57:Time to wake up task 1!
31    Running task: task1
```

4.4 Tasks

DSP/BIOS task objects are threads that are managed by the TSK module. Tasks have higher priority than the idle loop and lower priority than hardware and software interrupts.

The TSK module dynamically schedules and preempts tasks based on the task's priority level and the task's current execution state. This ensures that the processor is always given to the highest priority thread that is ready to run. There are 15 priority levels available for tasks. The lowest priority level (0) is reserved for running the idle loop.

The TSK module provides a set of functions that manipulate task objects. They access TSK object through handles of type TSK_Handle.

The kernel maintains a copy of the processor registers for each task object. Each task has its own run-time stack for storing local variables as well as for further nesting of function calls.

Stack size may be specified separately for each TSK object. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher priority task. If the task blocks, only those registers that a C function must save are saved to the task stack. To find the correct stack size, you can make the stack size large and then use Code Composer Studio to find the stack size actually used.

All tasks executing within a single program share a common set of global variables, accessed according to the standard rules of scope defined for C functions.

4.4.1 Creating Tasks

You can create TSK objects either dynamically (with a call to TSK_create) or statically (with the Configuration Tool). Tasks that you create dynamically can also be deleted during program execution.

4.4.1.1 Creating and Deleting Tasks Dynamically

You can spawn DSP/BIOS tasks by calling the function TSK_create(), whose parameters include the address of a C function in which the new task begins its execution. The value returned by TSK_create() is a handle of type TSK_Handle, which you can then pass as an argument to other TSK functions.

```
TSK_Handle TSK_create(fxn, attrs, [arg,] ...)
    Fxn        fxn;
    TSK_Attrs  *attrs
    Arg        arg
```

A task becomes active when it is created and preempts the currently running task if it has a higher priority.

The memory used by TSK objects and stacks may be reclaimed by calling TSK_delete(). TSK_delete() removes the task from all internal queues and frees the task object and stack by calling MEM_free().

Note that any semaphores, mailboxes, or other resources held by the task are *not* released. Deleting a task that holds such resources is often an application design error, although not necessarily so. In most cases, such resources should be released prior to deleting the task.

```
Void TSK_delete(task)
    TSK_Handle    task;
```

Note:

Catastrophic failure may occur if you delete a task that owns resources that are needed by other tasks in the system. See *TSK_delete*, in the *API Reference Guide* for details.

4.4.1.2 Creating Tasks with the Configuration Tool

You can also create tasks using the Configuration Tool. The Configuration Tool allows you to set a number of properties for each task and for the TSK Manager itself.

While it is running, a task that was created with the Configuration Tool behaves exactly the same as a task created with TSK_create. You cannot use the TSK_delete function to delete tasks created with the Configuration Tool. See section 2.2.4, *Creating Objects*, page 2-4, for a discussion of the benefits of creating objects with the Configuration Tool.

The default configuration template defines a TSK_smain task and a TSK_idle task:

❑ The TSK_main task is used during system startup and must have the highest priority. This task runs your program's smain function at the highest priority, so the main function must exit or be blocked after performing its initialization activities to allow other tasks to run.

❑ The TSK_idle task must have the lowest priority. It runs the functions defined for the IDL objects when no higher-priority task or interrupt is ready.

4.4.1.3 Setting Task Properties in the Configuration Tool

You can view the default TSK properties by clicking on the TSK Manager. Some of these properties include default task priority, stack size, and stack segment. Each time a new TSK object is inserted, its priority, stack size, and stack segment are set to the defaults. You can also set these properties individually for each TSK object. For a complete description of all TSK properties, see *TSK Module* in the *API Reference Guide*.

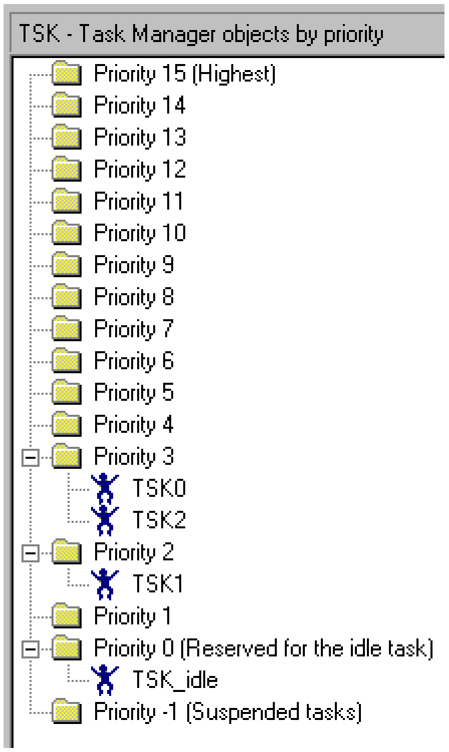
To change the priority of a task:

1) Open the TSK module in the Configuration Tool to view all statically created tasks.

2) If you select any task, you see its priority in the list of properties on the right side of the window.

3) To change the priority of a task object, drag the task to the folder of the corresponding priority. For example, to change the priority of TSK1 to 3, select it with the mouse and drag it to the folder labeled "Priority 3".

4) You can also change the priority of a task in the Properties window which you can select when you right-click on the TSK object pop-up menu



TSK - Task Manager objects by priority
- Priority 15 (Highest)
- Priority 14
- Priority 13
- Priority 12
- Priority 11
- Priority 10
- Priority 9
- Priority 8
- Priority 7
- Priority 6
- Priority 5
- Priority 4
- Priority 3
  - TSK0
  - TSK2
- Priority 2
  - TSK1
- Priority 1
- Priority 0 (Reserved for the idle task)
  - TSK_idle
- Priority -1 (Suspended tasks)

Notes:

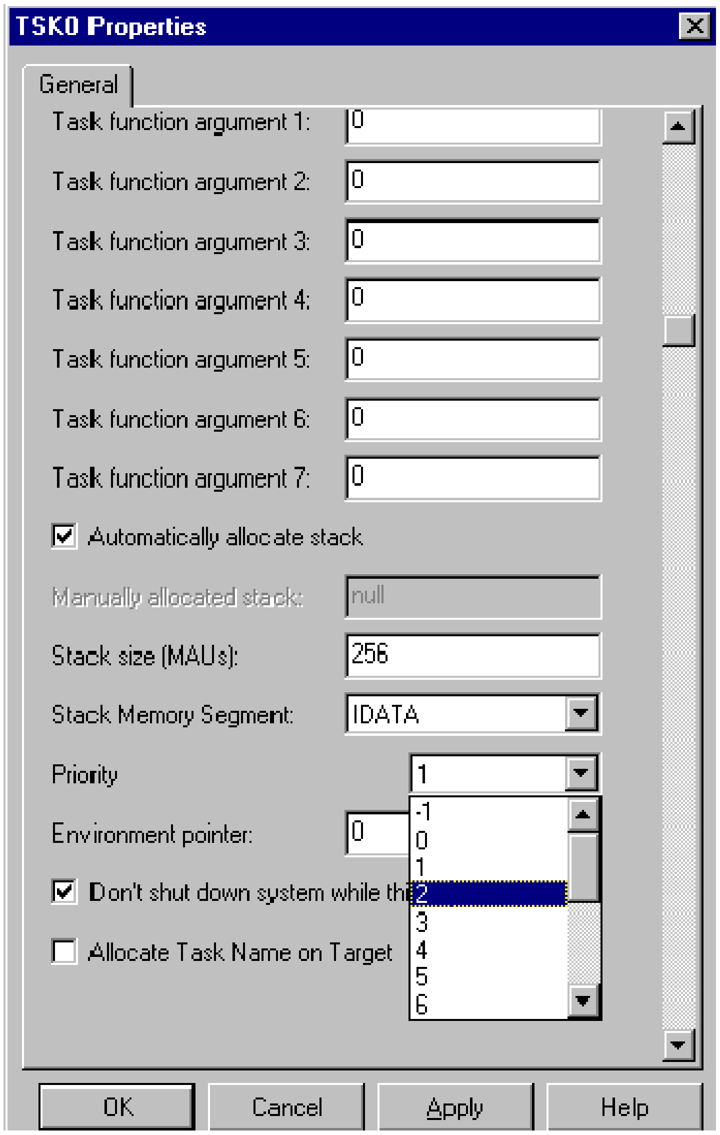
❑ When you use the Configuration Tool to create tasks of equal priority, they are scheduled in the order in which they are listed in the Configuration Tool window.

❑ Tasks can have up to 16 priority levels. The highest level is 15 and the lowest is 0. The priority level of 0 is reserved for the system idle task.

❑ If you want a task to be created in the suspended mode (TSK_BLOCKED), drag it to the folder labeled "Priority –1". For more information on task suspend, see section 4.4.2, *Task Execution States and Scheduling*, page 4-40.

❑ You cannot sort tasks within a single priority level.

- ❏ The Property window for a TSK object shows its numeric priority level (from 0 to 15; 15 is the highest level). You can also set the priority by selecting the priority level from the menu in the Property window.


TSK0 Properties

General

Task function argument 1:	0
Task function argument 2:	0
Task function argument 3:	0
Task function argument 4:	0
Task function argument 5:	0
Task function argument 6:	0
Task function argument 7:	0
☑ Automatically allocate stack	
Manually allocated stack:	null
Stack size (MAUs):	256
Stack Memory Segment:	IDATA
Priority	1
Environment pointer:	0
☑ Don't shut down system while th	
☐ Allocate Task Name on Target	

4.4.2 Task Execution States and Scheduling

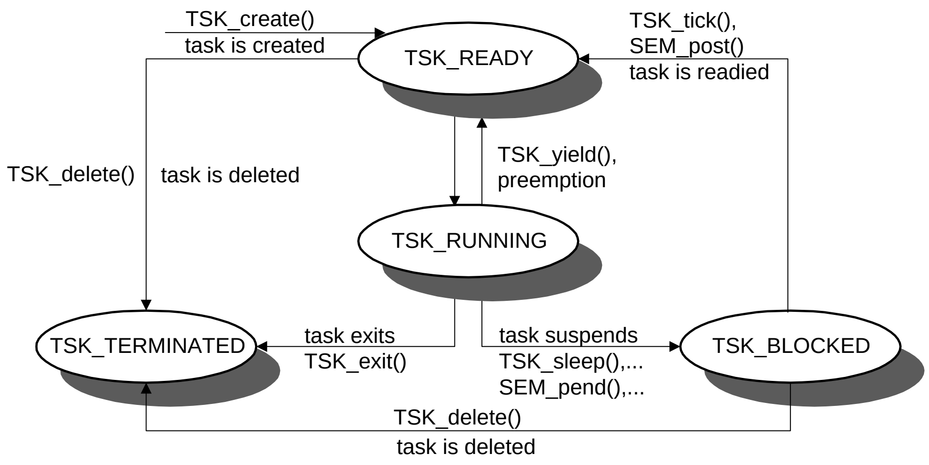
Each TSK task object is always in one of four possible states of execution:

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*, 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. There can be no more than one running task. As a rule, no ready task has a priority level greater than that of the currently running task, since TSK preempts the running task in favor of the higher-priority ready task. Unlike many time-sharing operating systems that give each task its "fair share" of the processor, DSP/BIOS *immediately* preempts the current task whenever a task of higher priority becomes ready to run.

The maximum priority level is TSK_MAXPRI (15); the minimum priority is TSK_MINPRI (1). If the priority is less than 0, the task is barred from further execution until its priority is raised at a later time by another task. If the priority equals TSK_MAXPRI, the task execution effectively locks out all other program activity except for the handling of hardware interrupts and software interrupts.

During the course of a program, each task's mode of execution can change for a number of reasons. The following figure shows how execution modes change:



Functions in the TSK, SEM, and SIO modules alter the execution state of task objects: blocking or terminating the currently running task, readying a previously suspended task, re-scheduling the current task, and so forth.

There is at exactly *one* task whose execution mode is TSK_RUNNING. If all program tasks are blocked and no hardware or software interrupt is running, TSK executes the TSK_idle task, whose priority is lower than all other tasks in the system. (When a task is preempted by a software or hardware interrupt, the task execution mode returned for that task by TSK_stat is still TSK_RUNNING because the task will run when the preemption ends.)

Note:

Do not make blocking calls, such as SEM_pend() or TSK_sleep(), from within an IDL function. Doing so prevents DSP/BIOS plug-ins from gathering run-time information.

When the TSK_RUNNING task transitions to any of the other three states, control switches to the highest-priority task that is ready to run (i.e., whose mode is TSK_READY). A TSK_RUNNING task transitions to one of the other modes in the following ways:

1) The running task becomes TSK_TERMINATED by calling TSK_exit(), which is automatically called if and when a task returns from its top-level

function. After all tasks have returned, the TSK manager terminates program execution by calling SYS_exit() with a status code of 0.

2) The running task becomes TSK_BLOCKED when it calls a function (for example, SEM_pend() or TSK_sleep()) that causes the current task to suspend its execution; tasks can move into this state when they are performing certain I/O operations, awaiting availability of some shared resource, or idling.

3) The running task becomes TSK_READY and is preempted whenever some other, higher-priority task becomes ready to run. TSK_setpri() can cause this type of transition if the priority of the current task is no longer the highest in the system. A task can also use TSK_yield() to yield to other tasks with the same priority. A task that yields becomes ready to run.

A task that is currently TSK_BLOCKED transitions to the ready state in response to a particular event: completion of an I/O operation, availability of a shared resource, the elapse of a specified period of time, and so forth. By virtue of becoming TSK_READY, this task is scheduled for execution according to its priority level; and, of course, this task immediately transitions to the running state if its priority is higher than the currently executing task. TSK schedules tasks of equal priority on a first-come, first-served basis.

4.4.3 Testing for Stack Overflow

When a task uses more memory than its stack has been allocated, it can write into an area of memory used by another task or data. This results in unpredictable and potentially fatal consequences. Therefore, a means of checking for stack overflow is useful.

Two functions, TSK_checkstacks(), and TSK_stat(), can be used to watch stack size. The structure returned by TSK_stat() contains both the size of its stack and the maximum number of MAUs ever used on its stack, so this code segment could be used to warn of a nearly full stack:

```
TSK_Stat statbuf;               /* declare buffer */

TSK_stat(TSK_self(), &statbuf); /* call func to get status */
if (statbuf.used > (statbuf.attrs.stacksize * 9 / 10)) {
   LOG_printf(&trace, "Over 90% of task’s stack is in use.\n")
}
```

See the *TSK_stat* and *TSK_checkstacks* in the *API Reference Guide*, for a description and examples of their use.

4.4.4 Task Hooks

System-specific functions may be called for each task create (TSK_create()), task delete (TSK_delete()), task exit (TSK_exit()), or context switch (TSK_sleep(), SEM_pend(), etc.). These functions may be used to extend a task's context beyond the basic processor register set. The user-definable Create function, Delete function, Ready function, and Exit function configuration properties are described in the reference section for the TSK module.

The Switch function is invoked during a task switch, if it is not set to the "no-operation" function, _SYS_nop. Within this function, the application can access both the current and next task handles.

```
Void (* Switch_function)
  (TSK_Handle curTask, TSK_Handle nexTask);
```

For example, this function can be used to save or restore additional task context (e.g., external hardware registers), to check for task stack overflow, or to monitor the time used by each task.

The function specified as the Switch function is called from within the kernel and may make only those function calls allowed from within a software interrupt handler (SWI handler). See section A.1, *Functions Callable by Tasks, SWI Handlers, or Hardware ISRs* in the *API Reference Guide* for a list of functions callable from an SWI handler.

The function specified as the Ready function is performed when a task becomes ready to run, even though the task might not be allowed to run until other tasks of equal or higher priority block, finish, or yield. This function might be used to examine the scheduling and execution of tasks. Like the Switch function, the Ready function is called from within the kernel and may only call functions allowed within a software interrupt handler (SWI handler).

There are no real constraints on what functions may be called within the Create function, Delete function, and Exit function since these are invoked *outside* the kernel.

You can set the task hook functions with the Configuration Tool by right-clicking on the TSK manager and selecting Properties from the pop-up menu. Note that functions written in C must have the leading "_" (underscore).

4.4.5 Example—Task Hooks for Extra Context

Consider, for example, a system that has special hardware registers (say, for extended addressing) that need to be preserved on a per task basis. The function doCreate() is used to allocate a buffer to maintain these registers on a per task basis, doDelete() is used to free this buffer, and doSwitch() is used to save and restore these registers.

Note that if task objects are created with the Configuration Tool, the Switch function should *not* assume (as this example does) that a task's environment is always set by the Create function.

```
#define CONTEXTSIZE     ‘size of additional context‘

Void doCreate(task)
    TSK_Handle       task;
{
    Ptr              context;

    context = MEM_alloc(0, CONTEXTSIZE, 0);
    TSK_setenv(task, context);    /* set task environment */
}

Void doDelete(task)
    TSK_Handle       task;
{
    Ptr              context;

    context = TSK_getenv(task);     /* get register buffer */
    MEM_free(0, context, CONTEXTSIZE);
}

Void doSwitch(from, to)
    TSK_Handle       from;
    TSK_Handle       to;
{
    Ptr              context;

    static Int first = TRUE;
    if (first) {
      first = FALSE;
      return;
    }

    context = TSK_getenv(from);      /* get register buffer */
    *context = ‘hardware registers‘; /* save registers */

    context = TSK_getenv(to);        /* get register buffer /
    ‘hardware registers‘ = *context; /* restore registers */
}
```

```
Void doExit(Void)
{
    TSK_Handle    usrHandle;
    /* get task handle, if needed */
    usrHandle = TSK_self();

    ‘perform user-defined exit steps‘
}
```

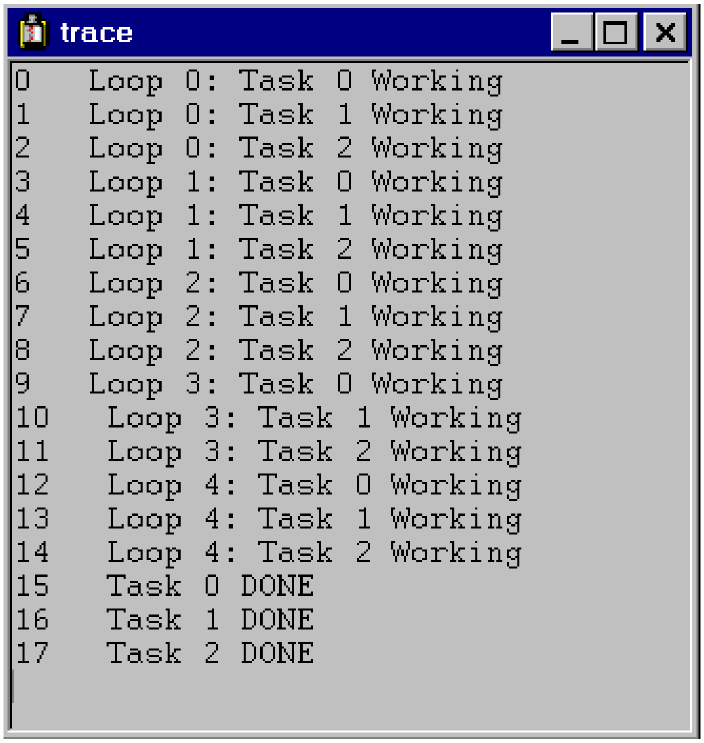
4.4.6 Example—Task Yielding for Round-Robin Scheduling

In the following programming example, three tasks of equal priority use TSK_yield() for “round-robin” scheduling. TSK_yield() effectively re-schedules the current task behind any other tasks of the same priority that are ready to run.

A listing of tsktest.c is shown next.

```
 /*
 *  ======== tsktest.c =======
 *  In this example, 3 tasks have been created with the
 *   Configuration Tool.  Each task has a computation loop
 *   consisting of a LOG_printf() followed by a
 *  TSK_yield().  This causes a round robin scheduling
 *  for these tasks of equal priority.
 */
#include <std.h>
#include <log.h>
#include <tsk.h>
#define NLOOPS  5

/*  Objects created with the Configuration Tool */
extern LOG_Obj trace;
Void task(Int id);
/*
 *  ======== main ========
 */
Void main()
{
}
/*
 *  ======== task ========
 */
Void task(Int id)
{
    Int     i;
       for (i = 0; i < NLOOPS ; i++) {
        LOG_printf(&trace, "Loop %d: Task %d Working", i, id);
        TSK_yield();
    }
    LOG_printf(&trace, "Task %d DONE", id);
}
```

This program should give you the following results:



```
0    Loop 0: Task 0 Working
1    Loop 0: Task 1 Working
2    Loop 0: Task 2 Working
3    Loop 1: Task 0 Working
4    Loop 1: Task 1 Working
5    Loop 1: Task 2 Working
6    Loop 2: Task 0 Working
7    Loop 2: Task 1 Working
8    Loop 2: Task 2 Working
9    Loop 3: Task 0 Working
10    Loop 3: Task 1 Working
11    Loop 3: Task 2 Working
12    Loop 4: Task 0 Working
13    Loop 4: Task 1 Working
14    Loop 4: Task 2 Working
15    Task 0 DONE
16    Task 1 DONE
17    Task 2 DONE
```

4.5 The Idle Loop

The idle loop is the background thread of DSP/BIOS, which runs continuously when no hardware interrupt service routines or software interrupts are running. Any other thread can preempt the idle loop at any point.

The IDL manager in the Configuration Tool allows you to insert functions that execute within the idle loop. The idle loop runs the IDL functions that you configured with the Configuration Tool. IDL_loop() calls the functions associated with each one of the IDL objects one at a time, and then starts over again in a continuous loop. The functions are called in the same order in which they were created in the Configuration Tool. Therefore, an IDL function must run to completion before the next IDL function can start running. When the last idle function has completed, the idle loop starts the first IDL function again. Idle loop functions are often used to poll non-real-time devices that do not (or cannot) generate interrupts, monitor system status, or perform other background activities.

The idle loop is the thread with lowest priority in a DSP/BIOS application. The idle loop functions run only when no other hardware or software interrupts need to run. Communication between the target and the DSP/BIOS plug-ins is performed within the background idle loop. This ensures that the DSP/BIOS plug-ins do not interfere with the program's tasks. If the target CPU is too busy to perform background tasks, the DSP/BIOS plug-ins stop receiving information from the target until the CPU is available.

By default, the idle loop runs the functions for these IDL objects:

- LNK_dataPump manages transfer of real-time analysis data (e.g., LOG and STS data), and HST channel data between the target DSP and the host. Different variants of LNK_dataPump support different target/host links; e.g., JTAG (via RTDX), shared memory, etc.
- RTA_dispatcher is a real-time analysis server on the target that accepts commands from DSP/BIOS plug-ins, gathers instrumentation information from the target, and uploads it at run time. RTA_dispatcher sits at the end of two dedicated HST channels; its commands/responses are routed from/to the host via LNK_dataPump.
- IDL_cpuLoad uses an STS object (IDL_busyObj) to calculate the target load. The contents of this object are uploaded to the DSP/BIOS plug-ins through RTA_dispatcher to display the CPU load.

4.6 Semaphores

DSP/BIOS provides a fundamental set of functions for inter-task synchronization and communication based upon *semaphores*.[1] Semaphores are often used to coordinate access to a shared resource among a set of competing tasks. The SEM module provides functions that manipulate semaphore objects accessed through handles of type SEM_Handle.

SEM objects are counting semaphores that may be used for both task synchronization and mutual exclusion. Counting semaphores keep an internal count of the number of corresponding resources available. When count is greater than 0, tasks do not block when acquiring a semaphore.

The functions SEM_create() and SEM_delete() are used to create and delete semaphore objects, respectively. You can also use the Configuration Tool to create semaphore objects. See section 2.2.7, *Creating and Deleting Objects Dynamically*, page 2-6, for a discussion of the benefits of creating objects with the Configuration Tool.

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.

```
SEM_Handle SEM_create(count, attrs);
    Uns        count;
    SEM_Attrs  *attrs;

Void SEM_delete(sem);
    SEM_Handle sem;
```

SEM_pend() waits for 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().

The timeout parameter to SEM_pend() allows the task to wait until a time-out, to wait indefinitely (SYS_FOREVER), or to not wait at all (0). SEM_pend()'s return value is used to indicate if the semaphore was acquired successfully.

```
Bool SEM_pend(sem, timeout);
  SEM_Handle sem;
  Uns   timeout; /* return after this many system clock ticks*/
```

SEM_post() is used to signal a semaphore. If a task is waiting for the semaphore, SEM_post() removes the task from the semaphore queue and

1. Most operating systems books contain additional information on the concept of semaphores.

puts it on the ready queue. If no tasks are waiting, SEM_post() simply increments the semaphore count and returns.

```
Void SEM_post(sem);
    SEM_Handle  sem;
```

4.6.1 SEM Example

In the following example three writer tasks create unique messages and place them on a queue. The writer tasks call SEM_post() to indicate that another message has been enqueued. The reader task calls SEM_pend() to wait for messages. SEM_pend() returns only when a message is available on the queue. The reader task prints the message using the LOG_printf() function.

The three writer tasks, reader task, semaphore, and queues in this example program were created with the Configuration Tool.

Since this program employs multiple tasks, a counting semaphore is used to synchronize access to the queue.

A listing of semtest.c is shown next.

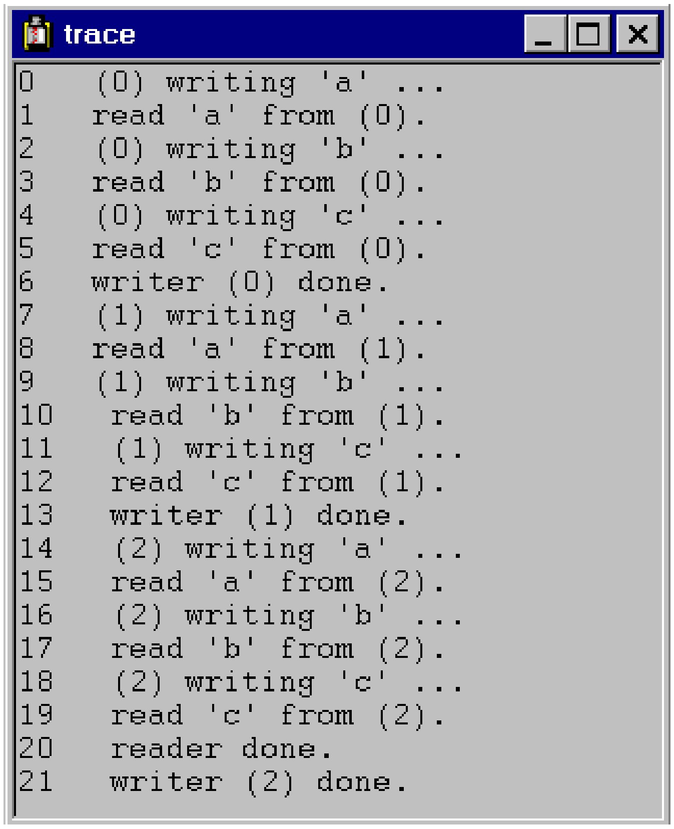
```
/*
 *  ======== semtest.c ========
 *
 *  Use a QUE queue and SEM semaphore to send messages from
 *  multiple writer() tasks to a single reader() task.  The
 *  reader task, the three writer tasks, queues, and semaphore
 *  are created by the Configuration Tool.
 *
 *  The MsgObj’s are preallocated in main(), and put on the
 *  free queue. The writer tasks get free message structures
 *  from the free queue, write the message, and then put the
 *  message structure onto the message queue.
 *  This example builds on quetest.c.  The major differences are:
 *    - one reader() and multiple writer() tasks.
 *    - SEM_pend() and SEM_post() are used to synchronize
 *      access to the message queue.
 *    - ‘id’ field was added to MsgObj to specify writer()
 *      task id.
 *
 *  Unlike a mailbox, a queue can hold an arbitrary number of
 *  messages or elements.  Each message must, however, be a
 *  structure with a QUE_Elem as its first field.
 */
```

```
#include <std.h>
#include <log.h>
#include <mem.h>
#include <que.h>
#include <sem.h>
#include <sys.h>
#include <tsk.h>
#include <trc.h>

#define NUMMSGS     3 /* number of messages */
#define NUMWRITERS  3 /* number of writer tasks created with */
                      /* Config Tool */

typedef struct MsgObj {
    QUE_Elem    elem;        /* first field for QUE */
    Int         id;          /* writer task id */
    Char        val;         /* message value */
} MsgObj, *Msg;

Void reader();
Void writer();

/*
 *  The following semaphore, queues, and log, are created by
 *  the Configuration Tool.
 */
extern SEM_Obj sem;

extern QUE_Obj msgQueue;
extern QUE_Obj freeQueue;

extern LOG_Obj trace;
```

```
/*
 *  ======== main ========
 */
Void main()
{
    Int        i;
    MsgObj     *msg;
    Uns        mask;

    mask = TRC_LOGTSK | TRC_LOGSWI | TRC_STSSWI | TRC_LOGCLK;
    TRC_enable(TRC_GBLHOST | TRC_GBLTARG | mask);

    msg = (MsgObj *)MEM_alloc(0, NUMMSGS * sizeof(MsgObj), 0);
    if (msg == MEM_ILLEGAL) {
        SYS_abort(“Memory allocation failed!\n”);
    }

    /* Put all messages on freequeue */
    for (i = 0; i < NUMMSGS; msg++, i++) {
        QUE_put(&freeQueue, msg);
    }
}

/*
 *  ======== reader ========
 */
Void reader()
{
    Msg         msg;
    Inti;

    for (i = 0; i < NUMMSGS * NUMWRITERS; i++) {
        /*
         * Wait for semaphore to be posted by writer().
         */
        SEM_pend(&sem, SYS_FOREVER);

        /* dequeue message */
        msg = QUE_get(&msgQueue);

        /* print value */
        LOG_printf(&trace, “read ‘%c’ from (%d).”, msg->val,
msg->id);

        /* free msg */
        QUE_put(&freeQueue, msg);
    }
    LOG_printf(&trace, “reader done.”);
}
```

```
/*
 *  ======== writer ========
 */
Void writer(Int id)
{
    Msg         msg;
    Int         i;

    for (i = 0; i < NUMMSGS; i++) {
        /*
         * Get msg from the free queue. Since reader is higher
         * priority and only blocks on sem, this queue is
         * never empty.
         */
        if (QUE_empty(&freeQueue)) {
            SYS_abort(“Empty free queue!\n”);
        }
        msg = QUE_get(&freeQueue);

        /* fill in value */
        msg->id = id;
        msg->val = (i & 0xf) + ‘a’;
        LOG_printf(&trace, “(%d) writing ‘%c’ ...”, id, msg-
>val);

        /* enqueue message */
        QUE_put(&msgQueue, msg);

        /* post semaphore */
        SEM_post(&sem);

        /* what happens if you call TSK_yield() here? */
        /* TSK_yield(); */
    }
    LOG_printf(&trace, “writer (%d) done.”, id);
}
```

This program should give you the following results:



```
0    (0) writing 'a' ...
1    read 'a' from (0).
2    (0) writing 'b' ...
3    read 'b' from (0).
4    (0) writing 'c' ...
5    read 'c' from (0).
6    writer (0) done.
7    (1) writing 'a' ...
8    read 'a' from (1).
9    (1) writing 'b' ...
10    read 'b' from (1).
11    (1) writing 'c' ...
12    read 'c' from (1).
13    writer (1) done.
14    (2) writing 'a' ...
15    read 'a' from (2).
16    (2) writing 'b' ...
17    read 'b' from (2).
18    (2) writing 'c' ...
19    read 'c' from (2).
20    reader done.
21    writer (2) done.
```

Though the three writer tasks are scheduled first, the messages are read as soon as they have been enqueued because the reader’s task priority is higher than that of the writer.

4.7 Mailboxes

The MBX module provides a set of functions to manage mailboxes. MBX mailboxes may be used to pass messages from one task to another on the same processor. An inter-task synchronization enforced by a fixed length shared mailbox can be used to ensure that the flow of incoming messages does not exceed the ability of the system to process those messages. The example given in this section illustrates just such a scheme.

The mailboxes managed by the MBX module are separate from the mailbox structure contained within a SWI object.

MBX_create() and MBX_delete() are used to create and delete mailboxes, respectively. You can also use the Configuration Tool to create mailbox objects. See section 2.2.7, *Creating and Deleting Objects Dynamically*, page 2-6, for a discussion of the benefits of creating objects with the Configuration Tool.

You specify the mailbox length and message size when you create a mailbox.

```
MBX_Handle MBX_create(msgsize, mbxlength, attrs)
    Uns        msgsize;
    Uns        mbxlength;
    MBX_Attrs  *attrs;

Void MBX_delete(mbx)
    MBX_Handle     mbx;
```

MBX_pend() is used to read a message from a mailbox. If no message is available (i.e., the mailbox is empty), MBX_pend() blocks. In this case, the timeout parameter allows the task to wait until a time-out, to wait indefinitely, or to not wait at all.

```
Bool MBX_pend(mbx, msg, timeout)
    MBX_Handle     mbx;
    Void           *msg;
    Uns            timeout;      /* return after this many */
                                 /* system clock ticks */
```

Conversely, MBX_post() is used to post a message to the mailbox. If no message slots are available (i.e., the mailbox is full), MBX_post() blocks. In this case, the timeout parameter allows the task to wait until a time-out, to wait indefinitely, or to not wait at all.

```
Bool MBX_post(mbx, msg, timeout)
    MBX_Handle     mbx;
    Void           *msg;
    Uns            timeout;      /* return after this many */
                                 /* system clock ticks */
```

4.7.1 MBX Example

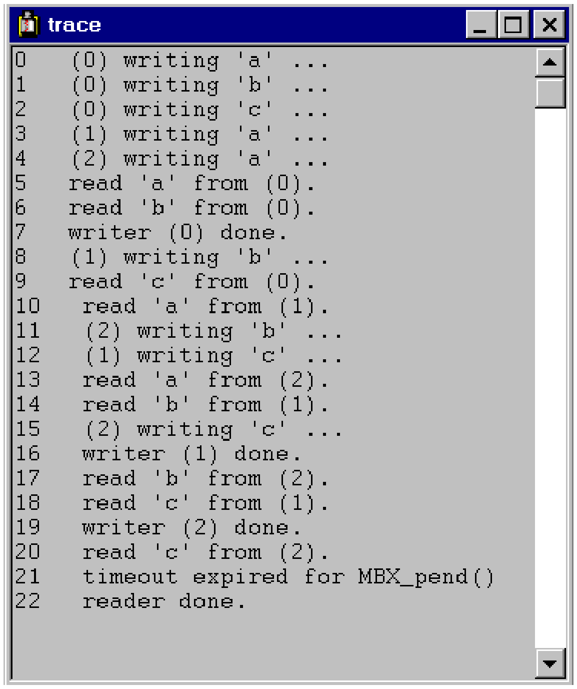
In the MBX example program below, there are two types of tasks created with the Configuration Tool: a single reader task which removes messages from the mailbox, and multiple writer tasks which insert messages into the mailbox.

A listing of mbxtest.c is shown next:

```
/*
 *  ======== mbxtest.c ========
 *  Use a MBX mailbox to send messages from multiple writer()
 *  tasks to a single reader() task.
 *  The mailbox, reader task, and 3 writer tasks are created
 *  by the Configuration Tool.
 *
 *  This example is similar to semtest.c. The major differences
 *  are:
 *  - MBX is used in place of QUE and SEM.
 *  - the 'elem' field is removed from MsgObj.
 *  - reader() task is *not* higher priority than writer task.
 *  - reader() looks at return value of MBX_pend() for timeout
 */

#include <std.h>

#include <log.h>
#include <mbx.h>
#include <tsk.h>

#define NUMMSGS         3       /* number of messages */
#define TIMEOUT         10

typedef struct MsgObj {
    Int    id;            /* writer task id */
    Char   val;           /* message value */
} MsgObj, *Msg;

/* Mailbox created with Config Tool */
extern MBX_Obj mbx;

/* "trace" Log created with Config Tool */
extern LOG_Obj trace;

Void reader(Void);
Void writer(Int id);

/*
 *  ======== main ========
 */
Void main()
{
    /* Does nothing */
}
```

```
/*
 *  ======== reader ========
 */
Void reader(Void)
{
    MsgObj      msg;
    Int         i;

    for (i=0; ;i++) {

        /* wait for mailbox to be posted by writer() */
        if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
          LOG_printf(&trace, "timeout expired for MBX_pend()");
            break;
        }

        /* print value */
        LOG_printf(&trace, "read ’%c’ from (%d).", msg.val,
msg.id);
    }
    LOG_printf(&trace, "reader done.");
}

/*
 *  ======== writer ========
 */
Void writer(Int id)
{
    MsgObj      msg;
    Int         i;

    for (i=0; i < NUMMSGS; i++) {
        /* fill in value */
        msg.id = id;
        msg.val = i % NUMMSGS + (Int)(‘a’);

        LOG_printf(&trace, "(%d) writing ‘%c’ ...", id,
(Int)msg.val);

        /* enqueue message */
        MBX_post(&mbx, &msg, TIMEOUT);

        /* what happens if you call TSK_yield() here? */
        /* TSK_yield(); */
    }
    LOG_printf(&trace, "writer (%d) done.", id);
}
```

After the program runs, review the trace log contents. The results should be similar to the following:



```
0   (0) writing 'a' ...
1   (0) writing 'b' ...
2   (0) writing 'c' ...
3   (1) writing 'a' ...
4   (2) writing 'a' ...
5   read 'a' from (0).
6   read 'b' from (0).
7   writer (0) done.
8   (1) writing 'b' ...
9   read 'c' from (0).
10   read 'a' from (1).
11   (2) writing 'b' ...
12   (1) writing 'c' ...
13   read 'a' from (2).
14   read 'b' from (1).
15   (2) writing 'c' ...
16   writer (1) done.
17   read 'b' from (2).
18   read 'c' from (1).
19   writer (2) done.
20   read 'c' from (2).
21   timeout expired for MBX_pend()
22   reader done.
```

Associated with the mailbox at creation time is a total number of available message slots, determined by the mailbox length you specify when you create the mailbox. In order to synchronize tasks writing to the mailbox, a counting semaphore is created and its count is set to the length of the mailbox. When a task does an MBX_post operation, this count is decremented. Another semaphore is created to synchronize the use of reader tasks with the mailbox; this counting semaphore is initially set to zero so that reader tasks block on empty mailboxes. When messages are posted to the mailbox, this semaphore is incremented.

In this example, all the tasks have the same priority. The writer tasks try to post all their messages, but a full mailbox causes each writer to block indefinitely. The readers then read the messages until they block on an empty mailbox. The cycle is repeated until the writers have exhausted their supply of messages.

At this point, the readers pend for a period of time according to the following formula, and then time out:

```
TIMEOUT*1ms/(clock ticks per millisecond)
```

After this timeout occurs, the pending reader task continues executing and then concludes.

At this point, it is a good idea to experiment with the relative effects of scheduling order and priority, the number of participants, the mailbox length, and the wait time by combining the following code modifications:

- ❏ creation order or priority of tasks;
- ❏ number of readers and writers;
- ❏ mailbox length parameter (MBXLENGTH); and/or
- ❏ add code to handle a writer task timeout.

4.8 Timers, Interrupts, and the System Clock

DSPs typically have one or more on-chip timers which generate a hardware interrupt at periodic intervals. DSP/BIOS normally uses one of the available on-chip timers as the source for its own system clock. Using the on-chip timer hardware present on most TMS320 DSPs, the CLK module supports time resolutions close to the single instruction cycle.

You define the system clock parameters in the DSP/BIOS configuration settings. In addition to defining parameters for the CLK module itself, you can define parameters for the CLK module's HWI object, since that object is pre-configured to use the HWI dispatcher. This allows you to manipulate the interrupt mask and cache control mask properties of the CLK ISR. In addition to the DSP/BIOS system clock, you can set up additional clock objects for invoking functions each time a timer interrupt occurs.

DSP/BIOS provides two separate timing methods—the high- and low-resolution times and the system clock. In the default configuration, the low-resolution time and the system clock are the same. However, your program can drive the system clock using some other event, such as the availability of data. You can disable or enable the CLK manager's use of the on-chip timer to drive high- and low-resolution times. You can drive the system clock using the low-resolution time, some other event, or not at all. The interactions between these two timing methods are as follows:

	CLK module drives system clock	**Other event drives system clock**	**No event drives system clock**
CLK manager enabled	Default configuration: Low-resolution time and system clock are the same	Low-resolution time and system clock are different	Only low- and high-resolution times available; timeouts don't elapse
CLK manager disabled	Not possible	Only system clock available; CLK functions don't run	No timing method; CLK functions don't run; timeouts don't elapse

4.8.1 High- and Low-Resolution Clocks

Using the CLK manager in the Configuration Tool, you can disable or enable DSP/BIOS' use of an on-chip timer to drive high- and low-resolution times. The TMS320C6000 has two general-purpose timers. The Configuration Tool allows you to select the on-chip timer that is used by the CLK manager. It also allows you to enter the period at which the timer interrupt is triggered. See *CLK Module* in the *API Reference Guide*, for more details about these properties. By entering the period of the timer interrupt, DSP/BIOS automatically sets up the appropriate value for the period register.

If the CLK manager is enabled in the Configuration Tool, the timer counter register is incremented every four CPU cycles.

When this register reaches the value set for the period register, the counter is reset to 0 and a timer interrupt occurs. When a timer interrupt occurs, the HWI object for the selected timer runs the CLK_F_isr function. This function causes these events to occur:

- The low-resolution time is incremented by 1.
- All the functions specified by CLK objects are performed in sequence in the context of that ISR.

Therefore, the low-resolution clock ticks at the timer interrupt rate and the clock's time is equal to the number of timer interrupts that have occurred. To obtain the low-resolution time, you can call CLK_getltime from your application code.

The CLK functions performed when a timer interrupt occurs are performed in the context of the hardware interrupt that caused the system clock to tick. Therefore, the amount of processing performed within CLK functions should be minimized and these functions may invoke only DSP/BIOS calls that are allowable from within a hardware ISR. (They should not call HWI_enter and HWI_exit as these are called internally by the HWI dispatcher when it runs CLK_F_isr.)

The high-resolution clock ticks at the same rate the timer counter register is incremented. Hence, the high-resolution time is the number of times the timer counter register has been incremented (or the number of instruction cycles divided by 4). Given the high CPU clock rate, the 32-bit timer counter register may reach the period register value very quickly. The 32-bit high-resolution time is actually calculated by multiplying the low-resolution time (i.e., the interrupt count) by the value of the period register and adding the current value of the timer counter register. To obtain the value of the high-resolution time you can call CLK_gethtime from your application code.

The values of both clocks restart at 0 when the maximum 32-bit value is reached.

Other CLK module APIs are CLK_getprd, which returns the value set for the period register in the Configuration Tool; and CLK_countspms, which returns the number of timer counter register increments per millisecond.

Modify the properties of the CLK manager with the Configuration Tool to configure the low-resolution clock. For example, to make the low-resolution clock tick every millisecond (.001 sec), type 1000 in the CLK manager's Microseconds/Int field. The Configuration Tool automatically calculates the correct value for the period register.

You can directly specify the period register value if you put a checkmark in the Directly configure on-chip timer registers box. To generate a 1 millisecond (.001 sec) system clock period on a 160 MIPS processor using the CPU clock/4 to drive the clock, the period register value is:

```
Period = 0.001 sec * 160,000,000 cycles per second / 4 cycles = 50,000
```

4.8.2 System Clock

Many DSP/BIOS functions have a timeout parameter. DSP/BIOS uses a “system clock” to determine when these timeouts should expire. The system clock tick rate can be driven using either the low-resolution time or an external source.

The TSK_sleep() function is an example of a function with a timeout parameter. After calling this function, its timeout expires when a number of ticks equal to the timeout value have passed in the system clock. For example, if the system clock has a resolution of 1 microsecond and we want the current task to block for 1 millisecond, the call should look like this:

```
/* block for 1000 ticks * 1 microsecond = 1 msec */
TSK_sleep(1000)
```

Note:

Do not call TSK_sleep() or SEM_pend() with a time-out other than 0 or SYS_FOREVER if the program is configured without something to drive the PRD module. In a default configuration, the CLK module drives the PRD module.

If you are using the default CLK configuration, the system clock has the same value as the low-resolution time because the PRD_clock CLK object drives the system clock.

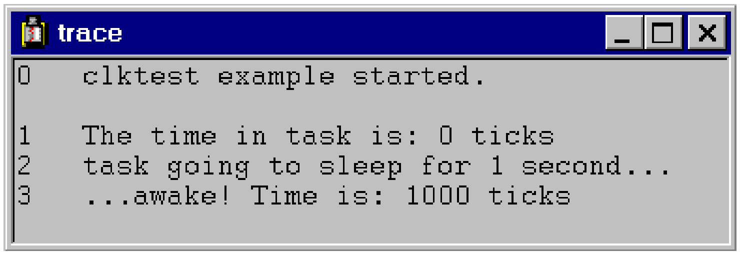
There is no requirement that an on-chip timer be used as the source of the system clock. An external clock, for example one driven by a data stream rate, can be used instead. If you do not want the on-chip timer to drive the low-resolution time, use the Configuration Tool to delete the CLK object named PRD_clock. If an external clock is used, it can call PRD_tick() to advance the system clock. Another possibility is having an on-chip peripheral such as the codec, that is triggering an interrupt at regular intervals, call PRD_tick() from that interrupt’s hardware ISR. In this case, the resolution of the system call is equal to the frequency of the interrupt from which PRD_tick() is called.

4.8.3 Example—System Clock

The following example, clktest.c, shows a simple use of the DSP/BIOS functions that use the system clock, TSK_time() and TSK_sleep(). The “task” task in clktest.c sleeps for 1000 ticks before it is awakened by the task scheduler. Since no other tasks have been created, the program runs the idle functions while the “task” is blocked. Note that the program assumes that the system clock is configured and driven by PRD_clock. This program is included in the c:\ti\c6000\examples\bios directory of the product.

```
/*  ======== clktest.c =======
 *  In this example, a task goes to sleep for 1 sec and
 *  prints the time after it wakes up. */

#include <std.h>

#include <log.h>
#include <clk.h>
#include <tsk.h>

extern LOG_Obj trace;

/*  ======== main ======== */
Void main()
{
    LOG_printf(&trace, "clktest example started.\n");
}

Void taskFxn()
{
    Uns ticks;

    LOG_printf(&trace, "The time in task is: %d ticks",
(Int)TSK_time());

    ticks = (1000 * CLK_countspms()) / CLK_getprd();

    LOG_printf(&trace, "task going to sleep for 1 second... ");
    TSK_sleep(ticks);
    LOG_printf(&trace, "...awake! Time is: %d ticks",
(Int)TSK_time());
}
```

The trace log output looks similar to this:



```
0   clktest example started.

1   The time in task is: 0 ticks
2   task going to sleep for 1 second...
3   ...awake! Time is: 1000 ticks
```

4.9 Periodic Function Manager (PRD) and the System Clock

Many applications need to schedule functions based on I/O availability or some other programmed event. Other applications can schedule functions based on a real-time clock.

The PRD manager allows you to create objects that schedule periodic execution of program functions. To drive the PRD module, DSP/BIOS provides a system clock. The system clock is a 32-bit counter that ticks every time PRD_tick is called. You can use the timer interrupt or some other periodic event to call PRD_tick and drive the system clock.

There can be several PRD objects but all are driven by the same system clock. The period of each PRD object determines the frequency at which its function is called. The period of each PRD object is specified in terms of the system clock time; i.e., in system clock ticks.

To schedule functions based on certain events, use the following procedures:

❑ **Based on a real-time clock**. Put a check mark in the Use CLK Manager to Drive PRD box by right-clicking on the PRD manager and selecting Properties in the Configuration Tool. By doing this you are setting the timer interrupt used by the CLK manager to drive the system clock. Note that when you do this a CLK object called PRD_clock is added to the CLK module. This object calls PRD_tick every time the timer interrupt goes off, advancing the system clock by one tick.

Note:

When the CLK manager is used to drive PRD, the system clock that drives PRD functions ticks at the same rate as the low-resolution clock. The low-resolution and system time coincide.

❑ **Based on I/O availability or some other event**. Remove the check mark from the Use the CLK Manager to Drive PRD box for the PRD manager. Your program should call PRD_tick to increment the system clock. In this case the resolution of the system clock equals the frequency of the interrupt from which PRD_tick is called.

4.9.1 Invoking Functions for PRD Objects

When PRD_tick is called two things occur:

❑ PRD_F_tick, the system clock counter, increases by one; i.e., the system clock ticks.

❑ An SWI called PRD_swi is posted.

Note that when a PRD object is created with the Configuration Tool, a new SWI object is automatically added called PRD_swi.

When PRD_swi runs, its function executes the following type of loop:

```
for ("Loop through period objects") {
   if ("time for a periodic function")
       "run that periodic function";
}
```

Hence, the execution of periodic functions is deferred to the context of PRD_swi, rather than being executed in the context of the ISR where PRD_tick was called. As a result, there may be a delay between the time the system tick occurs and the execution of the periodic objects whose periods have expired with the tick. If these functions run immediately after the tick, you should configure PRD_swi to have a high priority with respect to other threads in your application.

4.9.2 Interpreting PRD and SWI Statistics

Many tasks in a real-time system are periodic; that is, they execute continuously and at regular intervals. It is important that such tasks finish executing before it is time for them to run again. A failure to complete in this time represents a missed real-time deadline. While internal data buffering can be used to recover from occasional missed deadlines, repeated missed deadlines eventually result in an unrecoverable failure.

The implicit statistics gathered for SWI functions measure the time from when a software interrupt is ready to run and the time it completes. This timing is critical because the processor is actually executing numerous hardware and software interrupts. If a software interrupt is ready to execute but must wait too long for other software interrupts to complete, the real-time deadline is missed. Additionally, if a task starts executing, but is interrupted too many times for too long a period of time, the real-time deadline is also missed.

The maximum ready-to-complete time is a good measure of how close the system is to potential failure. The closer a software interrupt's maximum ready-to-complete time is to its period, the more likely it is that the system cannot survive occasional bursts of activity or temporary data-dependent increases in computational requirements. The maximum ready-to-complete time is also an indication of how much headroom exists for future product enhancements (which invariably require more MIPS).

Note:

DSP/BIOS does not implicitly measure the amount of time each software interrupt takes to execute. This measurement can be determined by running the software interrupt in isolation using either the simulator or the emulator to count the precise number of execution cycles required.

It is important to realize even when the sum of the MIPS requirements of all routines in a system is well below the MIPS rating of the DSP, the system may not meet its real-time deadlines. It is not uncommon for a system with a CPU load of less than 70% to miss its real-time deadlines due to prioritization problems. The maximum ready-to-complete times monitored by DSP/BIOS, however, provide an immediate indication when these situations exist.

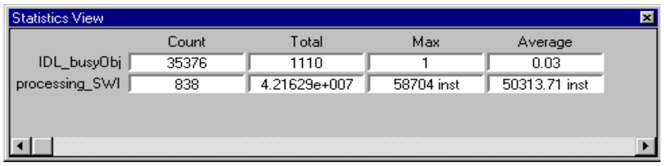
When statistics accumulators for software interrupts and periodic objects are enabled, the host automatically gathers the count, total, maximum, and average for the following types of statistics:

❑ **SWI**. Statistics about the period elapsed from the time the software interrupt was posted to its completion.

❑ **PRD**. The number of periodic system ticks elapsed from the time the periodic function is ready to run until its completion. By definition, a periodic function is ready to run when period ticks have occurred, where period is the ’period’ parameter for this object.

You can set the units for the SWI completion period by setting global SWI and CLK parameters. This period is measured in instruction cycles if the CLK module’s Use high resolution time for internal timings parameter is set to True (the default) and the SWI module’s Statistics Units parameter is set to Raw (the default). If this CLK parameter is set to False and the Statistics Units is set to Raw, SWI statistics are displayed in units of timer interrupt periods. You can also choose milliseconds or microseconds for Statistics Units.

For example, if the maximum value for a PRD object increases continuously, the object is probably not meeting its real-time deadline. In fact, the maximum value for a PRD object should be less than or equal to the period (in system ticks) property of this PRD object. If the maximum value is greater than the period, the periodic function has missed its real-time deadline.

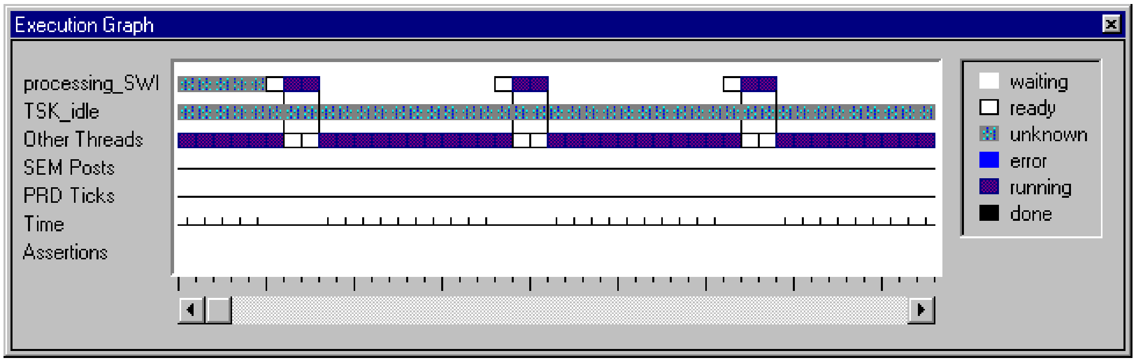


Statistics View

	Count	Total	Max	Average
IDL_busyObj	35376	1110	1	0.03
processing_SWI	838	4.21629e+007	58704 inst	50313.71 inst

4.10 Using the Execution Graph to View Program Execution

You can use the Execution Graph in Code Composer to see a visual display of thread activity by choosing Tools→DSP/BIOS→Execution Graph.



4.10.1 States in the Execution Graph Window

This window examines the information in the system log (LOG_system in the Configuration Tool) and shows the thread states in relation to the timer interrupt (Time) and system clock ticks (PRD Ticks).

White boxes indicate that a thread has been posted and is ready to run. Blue-green boxes indicate that the host had not yet received any information about the state of this thread at that point in the log. Dark blue boxes indicate that a thread is running.

Bright blue boxes in the Errors row indicate that an error has occurred. For example, an error is shown when the Execution Graph detects that a thread did not meet its real-time deadline. It also shows invalid log records, which may be caused by the program writing over the system log. Double-click on an error to see the details.

4.10.2 Threads in the Execution Graph Window

The SWI and PRD functions listed in the left column are listed from highest to lowest priority. However, for performance reasons, there is no information in the Execution Graph about hardware interrupt and background threads (aside from the CLK ticks, which are normally performed by an interrupt). Time not spent within an SWI, PRD, or TSK thread must be within an HWI or IDL thread, so this time is shown in the Other Threads row.

Functions run by PIP (notify functions) run as part of the thread that called the PIP API. The LNK_dataPump object runs a function that manages the host's end of an HST (Host Channel Manager) object. This object and other IDL objects run from the IDL background thread, and are included in the Other Threads row.

Note:

The Time marks and the PRD Ticks are not equally spaced. This graph shows a square for each event. If many events occur between two Time interrupts or PRD Ticks, the distance between the marks is wider than for intervals during which fewer events occurred.

4.10.3 Sequence Numbers in the Execution Graph Window

The tick marks below the bottom scroll bar show the sequence of events in the Execution Graph.

Note:

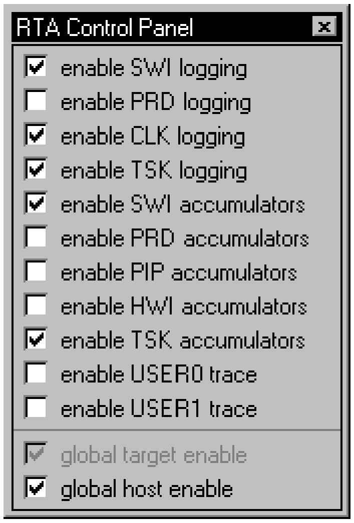
Circular logs (the default for the Execution Graph) contain only the most recent n events. Normally, there are events that are not listed in the log because they occur after the host polls the log and are overwritten before the next time the host polls the log. The Execution Graph shows a red vertical line and a break in the log sequence numbers at the end of each group of log events it polls.

You can view more log events by increasing the size of the log to hold the full sequence of events you want to examine. You can also set the RTA Control Panel to log only the events you want to examine.

4.10.4 RTA Control Panel Settings for Use with the Execution Graph

The TRC module allows you to control what events are recorded in the Execution Graph at any given time during the application execution. The recording of SWI, PRD, and CLK events in the Execution Graph can be controlled from the host (using the RTA Control Panel; Tools→DSP/BIOS→ RTA Control Panel in Code Composer) or from the target code (through the TRC_enable and TRC_disable APIs). See section 3.4.4.2, *Control of Implicit*

Instrumentation, page 3-16, for details on how to control implicit instrumentation.



RTA Control Panel

- [✓] enable SWI logging
- [ ] enable PRD logging
- [✓] enable CLK logging
- [✓] enable TSK logging
- [✓] enable SWI accumulators
- [ ] enable PRD accumulators
- [ ] enable PIP accumulators
- [ ] enable HWI accumulators
- [✓] enable TSK accumulators
- [ ] enable USER0 trace
- [ ] enable USER1 trace
- [✓] global target enable
- [✓] global host enable

When using the Execution Graph, turning off automatic polling stops the log from scrolling frequently and gives you time to examine the graph. You can use either of these methods to turn off automatic polling:

❑ Right-click on the Execution Graph and choose Pause from the shortcut menu.

❑ Right-click on the RTA Control Panel and choose Property Page. Set the Event Log/Execution Graph refresh rate to 0. Click OK.

You can poll log data from the target whenever you want to update the graph:

❑ Right-click on the Execution Graph and choose Refresh Window from the shortcut menu.

  You can choose Refresh Window several times to see additional data. The shortcut menu you see when you right-click on the graph also allows you to clear the previous data shown on the graph.

If you plan to use the Execution Graph and your program has a complex execution sequence, you can increase the size of the Execution Graph in the Configuration Tool. Right-click on the LOG_system LOG object and select Properties to increase the buflen property. Each log message uses four words, so the buflen should be at least the number of events you want to store multiplied by four.

Chapter 5

Memory and Low-level Functions

This chapter describes the low-level functions found in the DSP/BIOS real-time multitasking kernel. These functions are embodied in three software modules: MEM, which manages allocation of memory; SYS, which provides miscellaneous system services; and QUE, which manages queues.

This chapter also presents several simple example programs that use these modules. The system primitives are described in greater detail in Chapter 1, in the *API Reference Guide*.

Topic Page

5.1 Memory Management

The Memory Section Manager (MEM module) manages named memory sections that correspond to physical ranges of memory. If you want more control over memory sections, you can create your own linker command file and include the linker command file created by the Configuration Tool. It also provides a set of functions that can be used to dynamically allocate and free variable-sized blocks of memory.

Unlike standard C functions like malloc, the MEM functions enable you to specify which section of memory is used to satisfy a particular request for storage. Real-time DSP hardware platforms typically contain several different types of memory: fast, on-chip RAMs; zero wait-state external SRAMs; slower DRAMs for bulk data; and so forth.

Note:

Having explicit control over which memory section contains a particular block of data is essential to meeting real-time constraints in many DSP applications.

The MEM functions allocate and free variable-sized memory blocks. Memory allocation and freeing are non-deterministic when using the MEM module, since this module maintains a linked list of free blocks for each particular memory section. MEM_alloc and MEM_free must transverse this linked list when allocating and freeing memory.

5.1.1 Configuring Memory Sections

The templates provided with DSP/BIOS define a set of memory sections. These sections are somewhat different for each supported DSP board. If you are using a hardware platform for which there is no configuration template, you need to customize the MEM objects and their properties. You can customize MEM sections in the following ways:

- ❏ Insert a new MEM section and define its properties. For details on MEM object properties, see the *DSP/BIOS API Reference Guide*.
- ❏ Change the properties of an existing MEM section.
- ❏ Delete some MEM sections, particularly those that correspond to external memory locations. However, you must first change any references to that section made in the properties of other objects and managers. To find dependencies on a particular MEM section, right-click on that section and select Show Dependencies from the pop-up menu. Deleting or renaming the IPRAM and IDRAM sections is not recommended.

❏ Rename some MEM sections. To rename a section, follow these steps:

   a) Remove dependencies to the section you want to rename. To find dependencies on a particular MEM section, right-click on that section and select Show Dependencies from the pop-up menu.

   b) Rename the section. You can right-click on the section name and choose Rename from the pop-up menu to edit the name.

   c) Recreate dependencies on this section as needed by selecting the new section name in the property dialogs for other objects.

5.1.2 Disabling Dynamic Memory Allocation

If small code size is important to your application, you can reduce code size significantly by removing the capability to dynamically allocate and free memory. If you remove this capability, your program cannot call any of the MEM functions or any object creation functions (such as TSK_create). You should create all objects that are used by your program with the Configuration Tool.

To remove the dynamic memory allocation capability, put a checkmark in the No Dynamic Memory Heaps box in the Properties dialog for the MEM Manager.

If dynamic memory allocation is disabled and your program calls a MEM function (or indirectly calls a MEM function by calling an object creation function), an error occurs. If the segid passed to the MEM function is the name of a section, a link error occurs. If the segid passed to the MEM function is an integer, the MEM function will call SYS_error.

5.1.3 Defining Sections in Your Own Linker Command File

The MEM manager allows you to select which memory section contains various types of code and data. If you want more control over where these items are stored, put a checkmark in the User .cmd file for non-DSP/BIOS sections box in the Properties dialog for the MEM Manager.

You should then create your own linker command file that begins by including the linker command file created by the Configuration Tool. For example, your own linker command file might look like the following:

```
/* First include DSP/BIOS generated cmd file. */
-l designcfg.cmd

SECTIONS {
   /* place high-performance code in on-chip ram */
   .fast_text: {
      myfastcode.lib*(.text)
      myfastcode.lib*(.switch)
   } > IPRAM

   /* all other user code in off chip ram */
      .text:    {} > SDRAM0
      .switch:  {} > SDRAM0
      .cinit:   {} > SDRAM0
      .pinit:   {} > SDRAM0

   /* user data in on-chip ram */
      .bss:     {} > IDRAM
      .far:     {} > IDRAM
}
```

5.1.4 Allocating Memory Dynamically

Basic system-level storage allocation is handled by MEM_alloc, whose parameters specify a memory section, a block size, and an alignment. If the memory request cannot be satisfied, MEM_alloc returns MEM_ILLEGAL.

```
Ptr MEM_alloc(segid, size, align)
    Int segid;
    Uns size;
    Uns align;
```

The segid parameter identifies the memory section from which memory is to be allocated. This identifier may be an integer or a memory section name defined in the Configuration Tool. (The terms "memory section" and "memory segment" are used interchangeably in the DSP/BIOS properties and documentation.)

The memory block returned by MEM_alloc contains at least the number of minimum addressable units (MAUs) indicated by the size parameter. A minimum addressable unit for a processor is the smallest datum that can be loaded or stored in memory. An MAU may be viewed as the number of bits between consecutive memory addresses. The number of bits in an MAU varies with different DSP chips. The MAU for TMS320C6000 is an 8-bit byte.

The memory block returned by MEM_alloc starts at an address that is a multiple of the align parameter; no alignment constraints are imposed if align is 0. For example, an array of structures might be allocated as follows:

```
typedef struct Obj {
    Int    field1;
    Int    field2;
    Ptr    objArr;
} Obj;

objArr = MEM_alloc(SRAM, sizeof(Obj) * ARRAYLEN, 0);
```

Many DSP algorithms use circular buffers that can be managed more efficiently on most DSPs if they are aligned on a power of 2 boundary. This buffer alignment allows the code to take advantage of circular addressing modes found in many DSPs.

If no alignment is necessary, align should be 0. MEM's implementation aligns memory on a boundary equal to the number of words required to hold 64 bits, even if align has a value of 0. Other values of align cause the memory to be allocated on an align word boundary, where align is a power of 2.

5.1.5 Freeing Memory

MEM_free frees memory obtained with a previous call to MEM_alloc, MEM_calloc, or MEM_valloc. The parameters to MEM_free—segid, ptr, and size—specify a memory section, a pointer, and a block size respectively. Note that the values of these parameters must be the same as those used when allocating the block of storage.

```
Void MEM_free(segid, ptr, size)
    Int segid;
    Ptr ptr;
    Uns size;
```

As an example, the following function call frees the array of objects allocated in the previous example.

```
MEM_free(SRAM, objArr, sizeof(Obj) * ARRAYLEN);
```

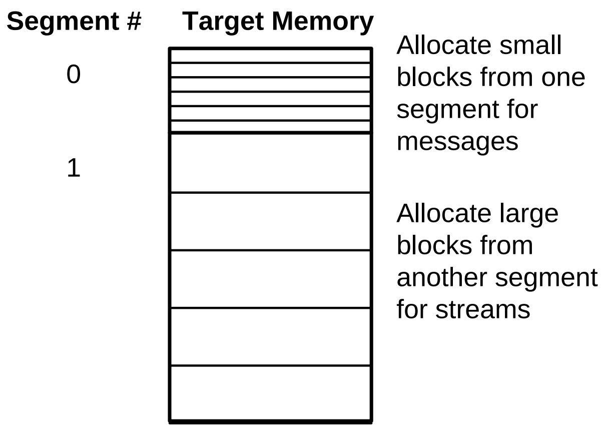
5.1.6 Getting the Status of a Memory Section

In a manner similar to MEM_alloc, MEM_calloc, and MEM_valloc, the size, used and length values returned by MEM_stat are the number of minimum addressable units (MAUs).

5.1.7 Reducing Memory Fragmentation

Repeatedly allocating and freeing blocks of memory can lead to memory fragmentation. When this occurs, calls to MEM_alloc can return MEM_ILLEGAL if there is no contiguous block of free memory large enough to satisfy the request. This occurs even if the total amount of memory in free memory blocks is greater than the amount requested.

To minimize memory fragmentation, you can use separate memory sections for allocations of different sizes as shown below.


Segment #
Target Memory
0
1
Allocate small blocks from one segment for messages
Allocate large blocks from another segment for streams

Note:

To minimize memory fragmentation, allocate smaller, equal-sized blocks of memory from one memory section and larger equal-sized blocks of memory from a second section.

5.1.8 MEM Example

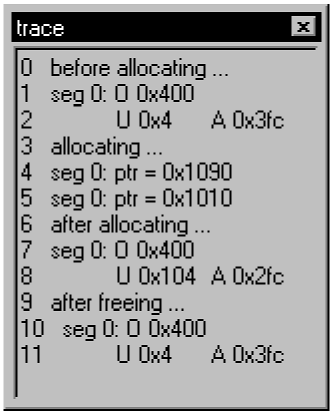
This example uses the functions MEM_stat, MEM_alloc, and MEM_free to highlight several issues involved with memory allocation.

A listing of memtest.c is shown next.

```
/*  ======== memtest.c ========
 *  This program allocates and frees memory from
 *  different memory segments.
 */

#include <std.h>
#include <log.h>
#include <mem.h>

#define NALLOCS 2      /* # of allocations from each segment */
#define BUFSIZE 128     /* size of allocations */

/* "trace" Log created by Configuration Tool */
extern LOG_Obj trace;
extern Int IDATA;
static Void printmem(Int segid);

/*
 *  ======== main ========
 */
Void main()
{
    Int i;
    Ptr ram[NALLOCS];
    LOG_printf(&trace, "before allocating ...");
    /* print initial memory status */
    printmem(IDRAM);
    LOG_printf(&trace, "allocating ...");
    /* allocate some memory from each segment */
    for (i = 0; i < NALLOCS; i++) {
        ram[i] = MEM_alloc(IDRAM, BUFSIZE, 0);
       LOG_printf(&trace, "seg %d: ptr = 0x%x", IDRAM, ram[i]);
    }
    LOG_printf(&trace, "after allocating ...");
    /* print memory status */
    printmem(IDRAM);
    /* free memory */
    for (i = 0; i < NALLOCS; i++) {
        MEM_free(IDRAM, ram[i], BUFSIZE);
    }
    LOG_printf(&trace, "after freeing ...");
    /* print memory status */
    printmem(IDRAM);
}
```

```
/*
 *  ======== printmem ========
 */
static Void printmem(Int segid)
{
    MEM_Stat    statbuf;
    MEM_stat(segid, &statbuf);
    LOG_printf(&trace, "seg %d: O 0x%x", segid, statbuf.size);
    LOG_printf(&trace, "\tU 0x%x\tA 0x%x", statbuf.used, stat-
buf.length);
}
```

This program gives board-dependent results. O indicates the original amount of memory, U the amount of memory used, and A the length in MAUs of the largest contiguous free block of memory. The addresses you see are likely to differ from the figure shown here.



```
0   before allocating ...
1   seg 0: O 0x400
2           U 0x4      A 0x3fc
3   allocating ...
4   seg 0: ptr = 0x1090
5   seg 0: ptr = 0x1010
6   after allocating ...
7   seg 0: O 0x400
8           U 0x104  A 0x2fc
9   after freeing ...
10  seg 0: O 0x400
11          U 0x4      A 0x3fc
```

In this example, memory is allocated from SRAM (external static RAM) and CRAM (on-chip RAM) memory using MEM_alloc, and later freed using MEM_free. printmem is used to print the memory status to the trace buffer. The final values (e.g., “after freeing...”) should match the initial values.

5.2 System Services

The SYS module provides a basic set of system services patterned after similar functions normally found in the standard C run-time library. As a rule, DSP/BIOS software modules use the services provided by SYS in lieu of similar C library functions.

Using the Configuration Tool, you can specify a customized routine that performs when the program calls one of these SYS functions. See the SYS reference section in the *API Reference Guide* for details.

5.2.1 Halting Execution

SYS provides two functions for halting program execution: SYS_exit, which is used for orderly termination; and SYS_abort, which is reserved for catastrophic situations. Since the actions that should be performed when exiting or aborting programs are inherently system-dependent, you can modify configuration settings to invoke your own routines whenever SYS_exit or SYS_abort is called.

```
Void SYS_exit(status)
    Int status;

Void SYS_abort(format, [arg,] ...)
    String  format;
    Arg arg;
```

These functions terminate execution by calling whatever routine is specified for the Exit function and Abort function properties of the SYS module. The default exit function is UTL_halt. The default abort function is _UTL_doAbort, which logs an error message and calls _halt. The _halt function is defined in the boot.c file; it loops infinitely with all processor interrupts disabled.

SYS_abort accepts a format string plus an optional set of data values (presumably representing a diagnostic message), which it passes to the function specified for the Abort function property of the SYS module as follows.

```
(*(Abort_function))(format, vargs)
```

The single vargs parameter is of type va_list and represents the sequence of arg parameters originally passed to SYS_abort. The function specified for the Abort function property may elect to pass the format and vargs parameters directly to SYS_vprintf or SYS_vsprintf prior to terminating program execution. To avoid the large code overhead of SYS_vprintf or SYS_vsprintf, you may want to use LOG_error instead to simply print the format string.

SYS_exit likewise calls whatever function is bound to the Exit function property, passing on its original status parameter. SYS_exit first executes a set of handlers registered through the function SYS_atexit as described below.

```
(*handlerN)(status)
        ...
(*handler2)(status)
(*handler1)(status)

(*(Exit_function))(status)
```

The function SYS_atexit provides a mechanism that enables you to stack up to SYS_NUMHANDLERS (which is set to 8) “clean-up” routines, which are executed before SYS_exit calls the function bound to the Exit function property. SYS_atexit returns FALSE when its internal stack is full.

```
Bool SYS_atexit(handler)
    Fxn    handler;
```

5.2.2 Handling Errors

SYS_error is used to handle DSP/BIOS error conditions. Application programs as well as internal functions use SYS_error to handle program errors.

```
Void SYS_error(s, errno, ...)
    String     s;
    Uns    errno;
```

SYS_error uses whatever function is bound to the Error function property to handle error conditions. The default error function in the configuration template is _UTL_doError, which logs an error message. For example, Error function may be configured to use doError which uses LOG_error to print the error number and associated error string.

```
Void doError(String s, Int errno, va_list ap)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = ’%s’", s);
}
```

The errno parameter to SYS_error may be a DSP/BIOS error (e.g., SYS_EALLOC) or a user error (errno >= 256). See Appendix A in the *API Reference Guide* for a table of error codes and strings.

5.3 Queues

The QUE module provides a set of functions to manage a list of QUE elements. Though elements can be inserted or deleted anywhere within the list, the QUE module is most often used to implement a FIFO list—elements are inserted at the tail of the list and removed from the head of the list.

QUE elements can be any structure whose first field is of type QUE_Elem.

```
typedef struct QUE_Elem {
    struct QUE_Elem *next;
    struct QUE_Elem *prev;
} QUE_Elem;
```

QUE_Elem is used by the QUE module to enqueue the structure while the remaining fields contain the actual data to be enqueued. For example, a structure used to enqueue a character might be declared as follows:

```
typedef struct MsgObj {
    QUE_Elem   elem;
    Char       val;
} MsgObj;
```

QUE_create and QUE_delete are used to create and delete queues, respectively. Since QUE queues are implemented as linked lists, queues have no maximum size.

```
QUE_Handle QUE_create(attrs)
    QUE_Attrs  *attrs;

Void QUE_delete(queue)
    QUE_Handle queue;
```

5.3.1 Atomic QUE Functions

QUE_put and QUE_get are used to atomically insert an element at the tail of the queue or remove an element from the head of the queue. These functions are atomic in that elements are inserted and removed with interrupts disabled. Therefore, multiple tasks (or tasks and ISRs) can safely use these two functions to modify a queue without any external synchronization.

QUE_get atomically removes and returns the element from the head of the queue. The return value has type Ptr to avoid unnecessary type casting by the calling function.

```
Ptr QUE_get(queue)
    QUE_Handle queue;
```

QUE_put atomically inserts the element at the tail of the queue. As with QUE_get, the queue element has type Ptr to avoid unnecessary type casting.

```
Ptr QUE_put(queue, elem)
    QUE_Handle queue;
    Ptr     elem;
```

5.3.2 Other QUE Functions

Unlike QUE_get and QUE_put, there are a number of QUE functions that do not disable interrupts when updating the queue. These functions must be used in conjunction with some mutual exclusion mechanism if the queues being modified are shared by multiple tasks (or tasks and ISRs).

QUE_dequeue and QUE_enqueue are equivalent to QUE_get and QUE_put except that they do not disable interrupts when updating the queue.

```
Ptr QUE_dequeue(queue)
    QUE_Handle queue;

Void QUE_enqueue(queue, elem)
    QUE_Handle queue;
    Ptr     elem;
```

QUE_head is used to return a pointer to the first element in the queue without removing the element. QUE_next and QUE_prev are used to scan the elements in the queue—QUE_next returns a pointer to the next element in the queue and QUE_prev returns a pointer to the previous element in the queue.

```
Ptr QUE_head(queue)
    QUE_Handle queue;

Ptr QUE_next(qelem)
    Ptr qelem;

Ptr QUE_prev(qelem)
    Ptr qelem;
```

QUE_insert and QUE_remove are used to insert or remove an element from an arbitrary point within the queue.

```
Void QUE_insert(qelem, elem)

    Ptr qelem;
    Ptr elem;

Void QUE_remove(qelem)
    Ptr qelem;
```

Note:

Since QUE queues are implemented as doubly linked lists with a header node, it is possible for QUE_head, QUE_next, or QUE_prev to return a pointer to the header node itself (e.g., calling QUE_head on an empty queue). Be careful not to call QUE_remove and remove this header node.

5.3.3 QUE Example

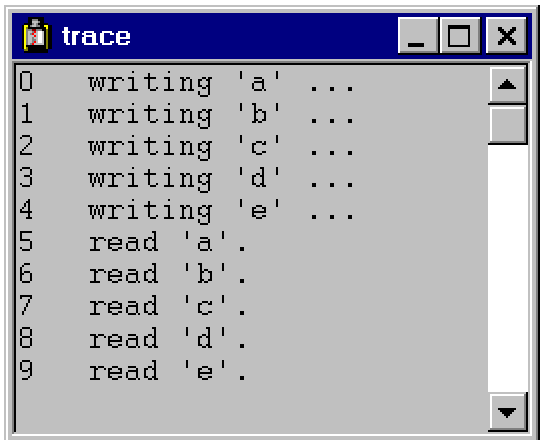
This example program uses a QUE queue to send five messages from a writer to a reader task. The functions MEM_alloc and MEM_free are used to allocate and free the MsgObj structures.

A listing of the example program quetest.c is shown next.

```
/*
 *  ======== quetest.c ========
 *  Use a QUE queue to send messages from a writer to a read
 *  reader.
 *
 *  The queue is created by the Configuration Tool.
 *  For simplicity, we use MEM_alloc and MEM_free to manage
 *  the MsgObj structures.  It would be way more efficient to
 *  preallocate a pool of MsgObj’s and keep them on a ’free’
 *  queue. Using the Config tool, create ’freeQueue’. Then in
 *  main, allocate the MsgObj’s with MEM_alloc and add them to
 *  ’freeQueue’ with QUE_put.
 *  You can then replace MEM_alloc calls with QUE_get(freeQueue)
 *  and MEM_free with QUE_put(freeQueue, msg).
 *
 *  A queue can hold an arbitrary number of messages or elements.
 *  Each message must, however, be a structure with a QUE_Elem as
 *  its first field.
 */

#include <std.h>
#include <log.h>
#include <mem.h>
#include <que.h>
#include <sys.h>

#define NUMMSGS     5      /* number of messages */

typedef struct MsgObj {
    QUE_Elem    elem;      /* first field for QUE */
    Char        val;       /* message value */
} MsgObj, *Msg;

extern QUE_Obj queue;

/* Trace Log created with the Configuration Tool */
extern LOG_Obj trace;

Void reader();
Void writer();
```

```
/*
 *  ======== main ========
 */
Void main()
{
    /*
     *  Writer must be called before reader to ensure that the
     *  queue is non-empty for the reader.
     */
    writer();
    reader();
}

/*
 *  ======== reader ========
 */
Void reader()
{
    Msg         msg;
    Int         i;
    for (i=0; i < NUMMSGS; i++) {
        /* The queue should never be empty */
        if (QUE_empty(&queue)) {
            SYS_abort("queue error\n");
        }
        /* dequeue message */
        msg = QUE_get(&queue);

        /* print value */
        LOG_printf(&trace, "read ’%c’.", msg->val);
        /* free msg */
        MEM_free(0, msg, sizeof(MsgObj));
    }
}

/*
 *  ======== writer ========
 */
Void writer()
{
    Msg         msg;
    Int         i;
    for (i=0; i < NUMMSGS; i++) {
        /* allocate msg */
        msg = MEM_alloc(0, sizeof(MsgObj), 0);
        if (msg == MEM_ILLEGAL) {
            SYS_abort("Memory allocation failed!\n");
        }
        /* fill in value */
        msg->val = i + ’a’;
        LOG_printf(&trace, "writing ’%c’ ...", msg->val);
        /* enqueue message */
        QUE_put(&queue, msg);
    }
}
```

This program yields the following results. The writer task uses QUE_put to enqueue each of its five messages and then the reader task uses QUE_get to dequeue each message.



```
0    writing 'a' ...
1    writing 'b' ...
2    writing 'c' ...
3    writing 'd' ...
4    writing 'e' ...
5    read 'a'.
6    read 'b'.
7    read 'c'.
8    read 'd'.
9    read 'e'.
```

Chapter 6

Kernel Aware Debug

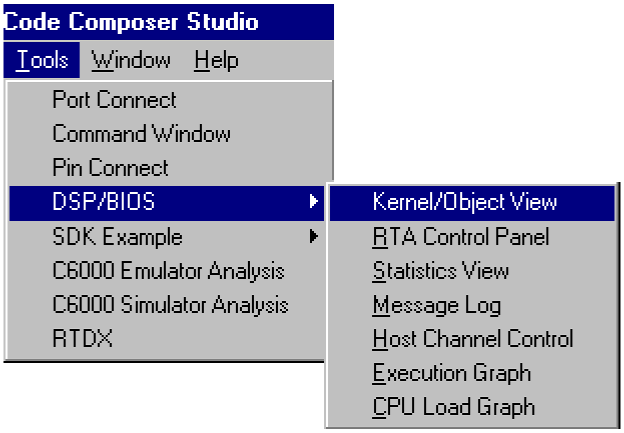
The Kernel Object View debug tool allows a view into the current configuration, state and status of the DSP/BIOS objects currently running on the target.

Topic **Page**

6.1 Debugging DSP/BIOS Applications

6.1.1 Kernel/Object View Debugger

The Kernel/Object View debug tool allows a view into the current configuration, state, and status of the DSP/BIOS objects currently running on the target. To start Kernel/Object View in Code Composer Studio, go to "Tools->DSP/BIOS->Kernel/Object View" as shown below.



There are six other pages of object data available to you: Tasks, Mailboxes, Semaphores, Memory, and Software Interrupts. The following sections describe these pages.

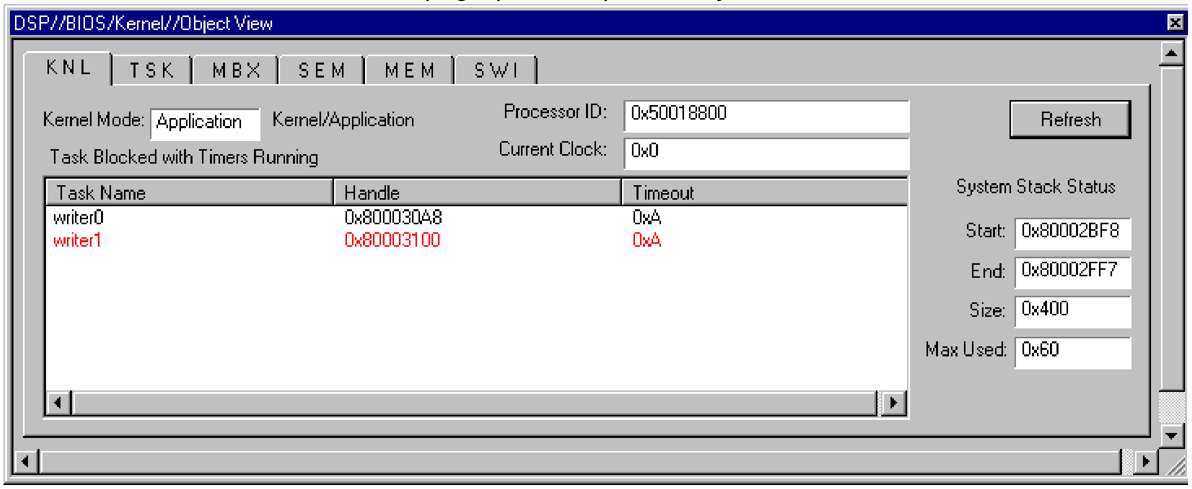
All pages have a "Refresh" button in the upper right corner. When this button is clicked on any page, it updates all the pages concurrently so that the data remains consistent on all pages. If the refresh button is pressed while the target is running, the target is halted, the data is collected, and then the target is restarted. All changes in displayed data are indicated by red text. If a stack overflow is detected, the data field containing the peak used value turns red, and yellow text flags the error.

Note:

The Kernel/Object View may display names that are labels for other items on the target because some labels share memory locations. In this case you may see a name that does not match the configuration. If a label is not available for a memory location, a name is automatically generated and is indicated with angle brackets (e.g., <task1>).

6.2 Kernel

The kernel page (tab: KNL) shows system-wide information.



Kernel Mode: Application Kernel/Application

Processor ID: 0x50018800

Current Clock: 0x0

Task Blocked with Timers Running

Task Name	Handle	Timeout
writer0	0x800030A8	0xA
writer1	0x80003100	0xA

System Stack Status

Start: 0x80002BF8

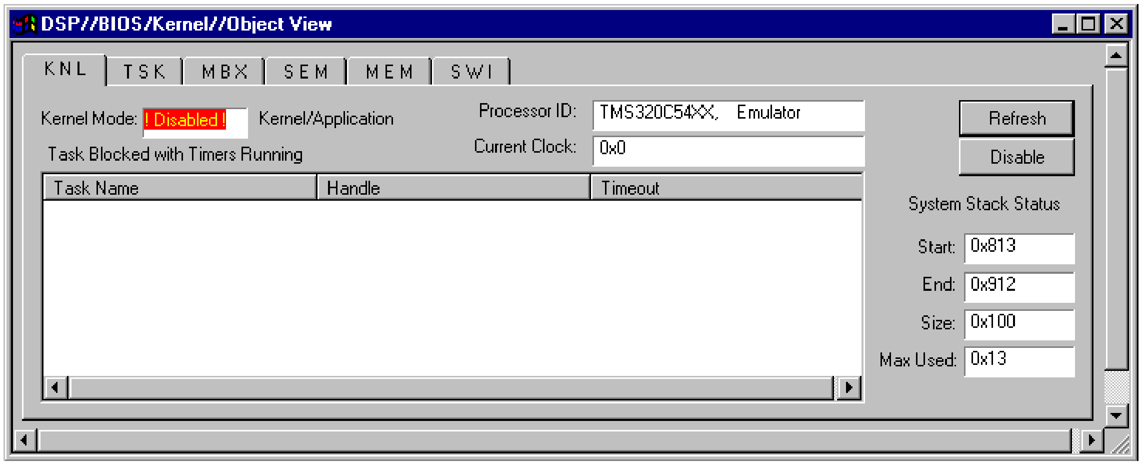
End: 0x80002FF7

Size: 0x400

Max Used: 0x60

- **Kernel Mode**: The value in this field indicates the current operating mode of the target. When the text "Kernel" appears, it indicates that the program is currently executing inside DSP/BIOS while the text "Application" indicates that the application is executing.
- **Processor ID**: The Processor ID field indicates the target processor and whether it is an emulator or simulator.
- **Current Clock:** This is the current value of the clock that is used for timer functions and alarms. The clock is set up during configuration (PRD_clk) in CLK - Clock Manager.
- **System Stack Status**: The four boxes on the right edge indicate system stack information.
- **Tasks Blocked with Timers Running**: This list contains all tasks that are currently blocked and have timers running to unblock them in the case that they are not made ready by any other means (they get the semaphore, a message in a mailbox and so on).
- **Disable:** The disable button allows you to disable the Kernel/Object View tool while you single-step through code or run through multiple break points. Since the tool takes some time to read all of the kernel data, you may want to disable it on occasion to step through to some critical code. The tool can only be disabled from the kernel page and is enabled by

pressing the refresh button or by changing pages to another object view. When the tool is disabled, Kernel Mode is set to "!Disabled!".



DSP//BIOS/Kernel//Object View

Kernel Mode: ! Disabled ! Kernel/Application

Processor ID: TMS320C54XX, Emulator

Current Clock: 0x0

Task Blocked with Timers Running

Task Name	Handle	Timeout

System Stack Status

Start: 0x813

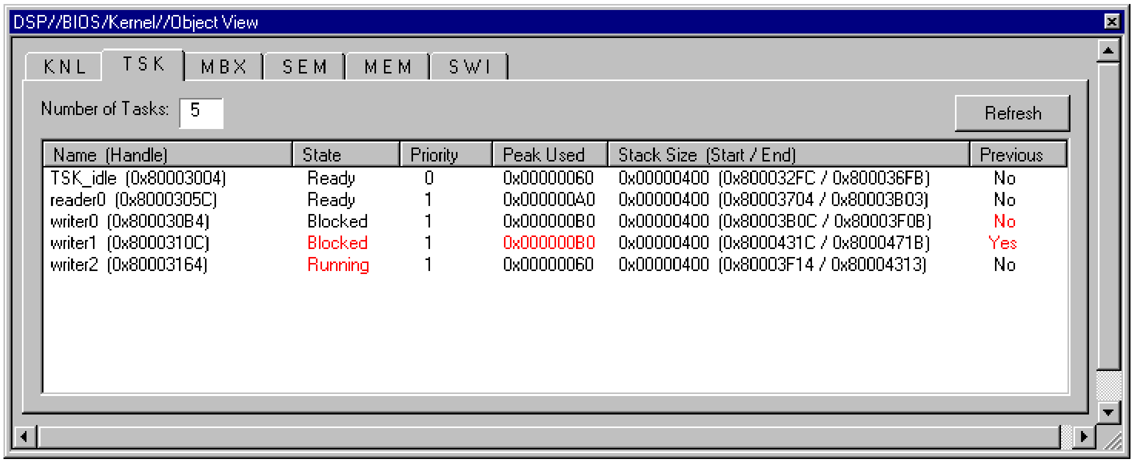
End: 0x912

Size: 0x100

Max Used: 0x13

6.3 Tasks

The tasks page ("TSK" on the tab) shows all task information.



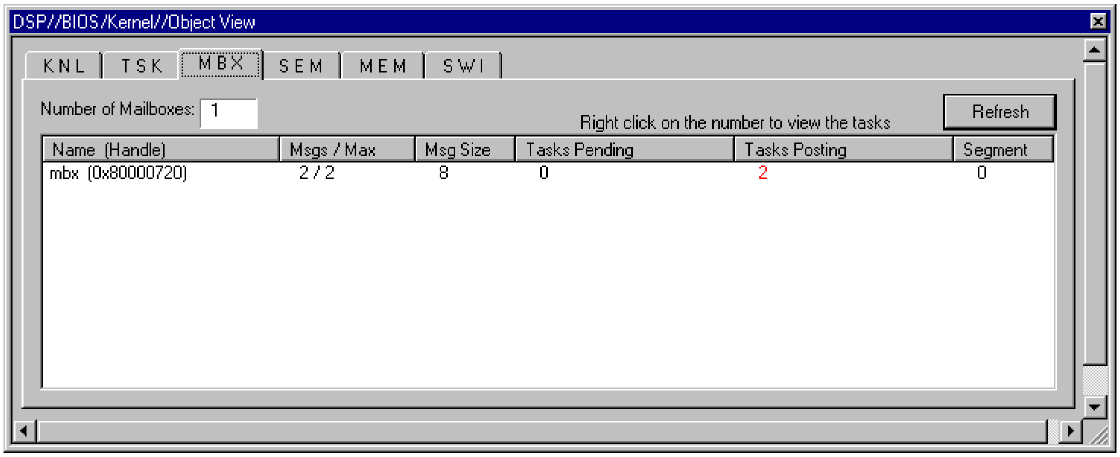
Number of Tasks: 5

Name (Handle)	State	Priority	Peak Used	Stack Size (Start / End)	Previous
TSK_idle (0x80003004)	Ready	0	0x00000060	0x00000400 (0x800032FC / 0x800036FB)	No
reader0 (0x8000305C)	Ready	1	0x000000A0	0x00000400 (0x80003704 / 0x80003B03)	No
writer0 (0x800030B4)	Blocked	1	0x000000B0	0x00000400 (0x80003B0C / 0x80003F0B)	No
writer1 (0x8000310C)	Blocked	1	0x000000B0	0x00000400 (0x8000431C / 0x8000471B)	Yes
writer2 (0x80003164)	Running	1	0x00000060	0x00000400 (0x80003F14 / 0x80004313)	No

- **Name (Handle):** This is the task name and handle. The name is taken from the label for statically configured objects and is generated for dynamically created objects. The label matches the name in the task manager configuration. The handle is the address on the target.
- **State**: The current state of the task: Ready, Running, Blocked, or Terminated.
- **Priority**: This is the task's priority as set in the configuration or as set by the API. Valid priorities are 0 through 15.
- **Peak Used**: This is the peak stack usage for the task. Since it shows the maximum ever used by the task a warning will appear if this value ever matches the Stack Size value in the next column. A warning is indicated when this field is red and the text is yellow.
- **Stack Size (Start/End)**: This is the stack size and the beginning and end of the stack in memory.
- **Previous**: "Yes" indicates that this task was the one running before the current task started.

6.4 Mailboxes

The mailboxes page ("MBX" on the tab) shows all mailbox information.



Number of Mailboxes: 1

Right click on the number to view the tasks

Name (Handle)	Msgs / Max	Msg Size	Tasks Pending	Tasks Posting	Segment
mbx (0x80000720)	2 / 2	8	0	2	0

❑ **Name (Handle)**: This is the mailbox name and handle. The name is taken from the label for statically configured objects and is generated for dynamically created objects. The label matches the name in the mailbox manager configuration. The handle is the address on the target.

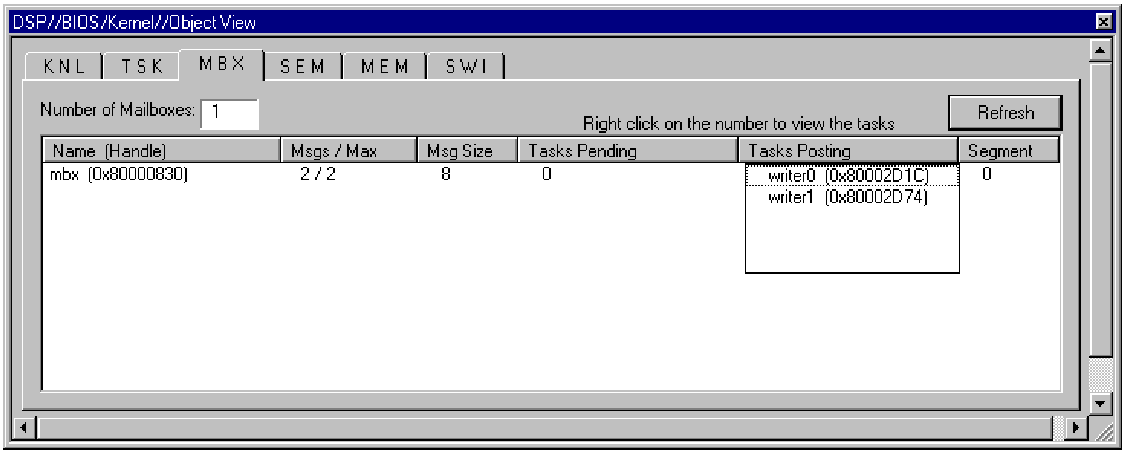
❑ **Msgs / Max**: The first number is the current number of messages that the mailbox contains. The second number is the maximum number of messages that the mailbox can hold. The maximum will match the value set in the configuration.

❑ **Msg Size**: This is the size of each message in the processor's minimum adressable units (MAUs). This matches the values set during configuration or creation.

❑ **Segment**: This is the memory segment number.

❑ **Tasks Pending**: This is the number of tasks currently blocked waiting to read a message from the mailbox. If the value is non-zero you may click on the number to see the list of tasks.

- **Tasks Posting**: This is the number of tasks currently blocked waiting to write a message to the mailbox. If the value is non-zero you may click on the number to see the list of tasks.



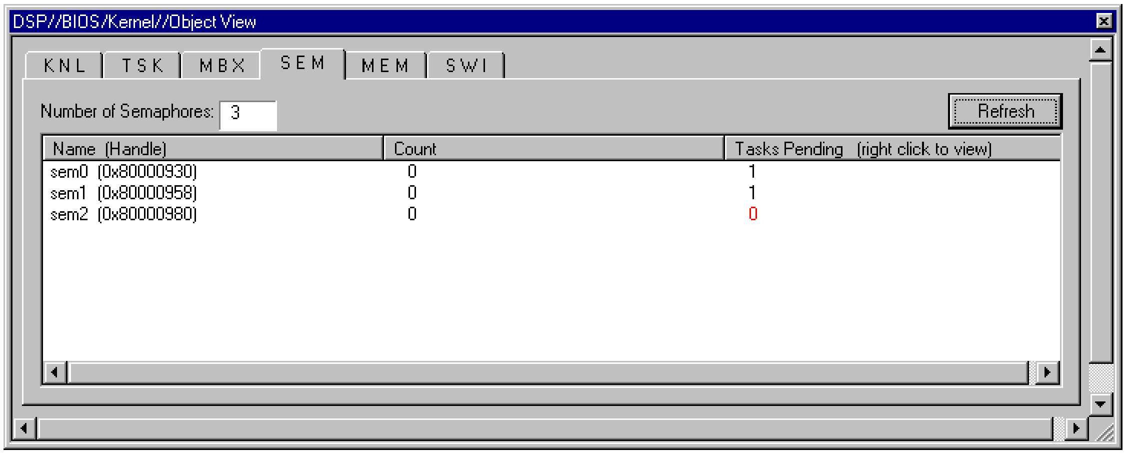
Number of Mailboxes: 1

Right click on the number to view the tasks

Name (Handle)	Msgs / Max	Msg Size	Tasks Pending	Tasks Posting	Segment
mbx (0x80000830)	2 / 2	8	0	writer0 (0x80002D1C) writer1 (0x80002D74)	0

6.5 Semaphores

The semaphores page ("SEM" on the tab) shows all semaphore information.



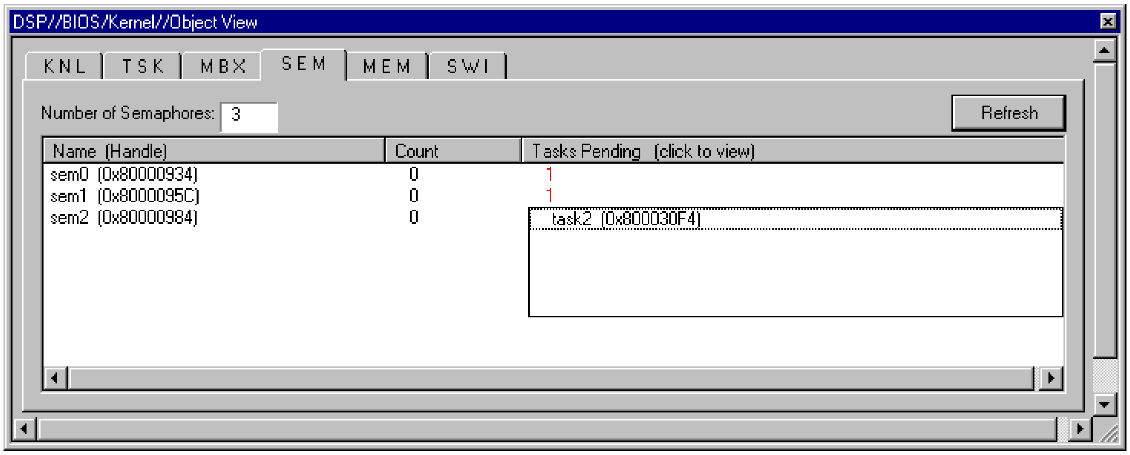
Number of Semaphores: 3

Name (Handle)	Count	Tasks Pending (right click to view)
sem0 (0x80000930)	0	1
sem1 (0x80000958)	0	1
sem2 (0x80000980)	0	0

❑ **Name (Handle)**: This is the semaphore name and handle. The name is taken from the label for statically configured objects and is generated for dynamically created objects. The label matches the name in the semaphore manager configuration. The handle is the address on the target.

❑ **Count**: This is the current semaphore count. This is the number of "pends" that can occur before blocking.

❑ **Tasks Pending**: This is the current number of tasks pending on the semaphore. If the value is non-zero you may click on the number to see a list of tasks that are pending.

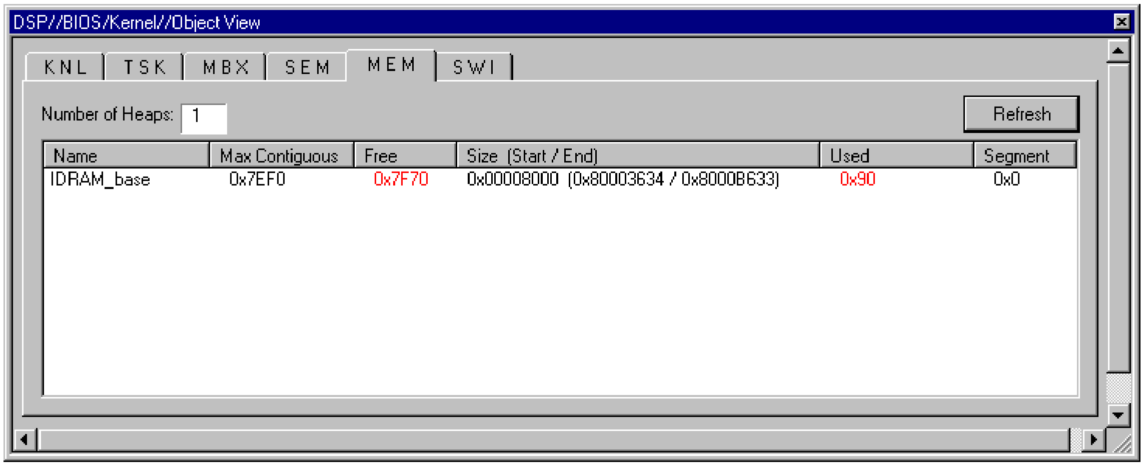


Number of Semaphores: 3

Name (Handle)	Count	Tasks Pending (click to view)
sem0 (0x80000934)	0	1
sem1 (0x8000095C)	0	1
sem2 (0x80000984)	0	task2 (0x800030F4)

6.6 Memory

The memory page ("MEM" on the tab) shows all memory heap information.



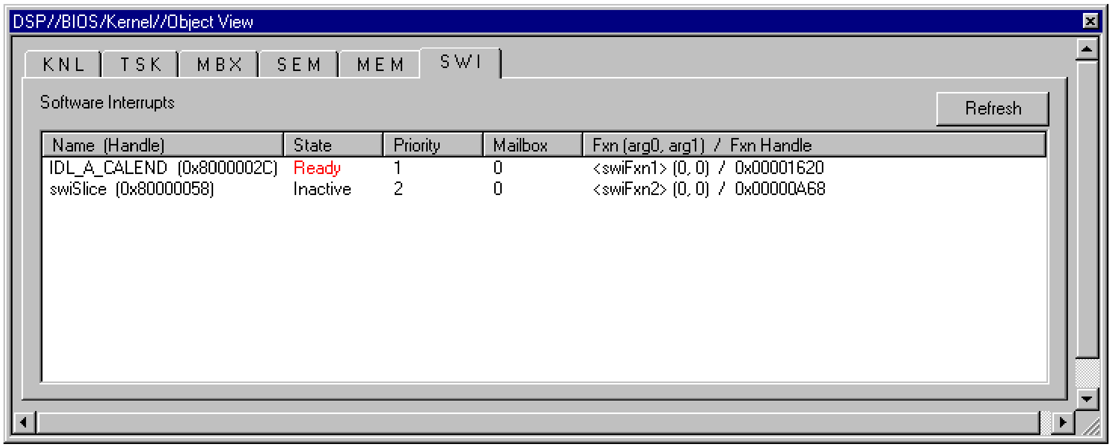
Number of Heaps: 1

Name	Max Contiguous	Free	Size (Start / End)	Used	Segment
IDRAM_base	0x7EF0	0x7F70	0x00008000 (0x80003634 / 0x8000B633)	0x90	0x0

- **Name**: This is the memory section that the heap is allocated from as configured.
- **Max Contiguous**: This is the maximum amount of contiguous memory that is free to allocate in the heap.
- **Free**: This is the total amount of memory that is free to allocate in the heap. If this value is zero a warning will be given. The warning indication will turn the field red and the letters yellow.
- **Size (Start / End)**: This is the heap size and the starting and ending locations in memory.
- **Used**: This is the amount of memory that is allocated from the heap. If this value is equal to the size, a warning will be given. The warning indication will turn the field red and the text yellow.
- **Segment**: This is the memory segment.

6.7 Software Interrupts

The software interrupts page ("SWI" on the tab) shows all software interrupt information.



Name (Handle)	State	Priority	Mailbox	Fxn (arg0, arg1) / Fxn Handle
IDL_A_CALEND (0x8000002C)	Ready	1	0	<swiFxn1> (0, 0) / 0x00001620
swiSlice (0x80000058)	Inactive	2	0	<swiFxn2> (0, 0) / 0x00000A68

Note:

The two function names are generated in this case as indicated by the angle brackets that surround the name.

- ❑ **Name (Handle):** This is the software interrupt name and handle. The name is taken from the label for statically configured objects and is generated for dynamically created objects. The label matches the name in the software interrupt manager configuration. The handle is the address on the target.
- ❑ **State**: This is the software interrupt's current state. Valid states are Inactive, Ready, or Running.
- ❑ **Priority**: This is the software interrupt's priority as set in the configuration or during creation. Valid priorities are 0 through 15.
- ❑ **Mailbox**: This is the software interrupt's current mailbox value.
- ❑ **Fxn (arg0, arg1) / Fxn Handle**: This is the software interrupt's function and arguments as set in the configuration or during creation. The handle is the address on the target.

Chapter 7

Input/Output Overview and Pipes

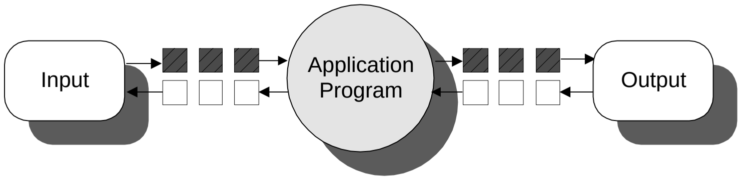
This chapter provides an overview on data transfer methods, and discusses pipes in particular.

Topic **Page**

7.1 I/O Overview

Input and output for DSP/BIOS applications are handled by stream, pipe, and host channel objects. Each type of object has its own module for managing data input and output.

A stream is a channel through which data flows between an application program and an I/O device. This channel can be read-only (input) or write-only (output) as shown in the figure below. Streams provide a simple and universal interface to all I/O devices, allowing the application to be completely ignorant of the details of an individual device's operation.



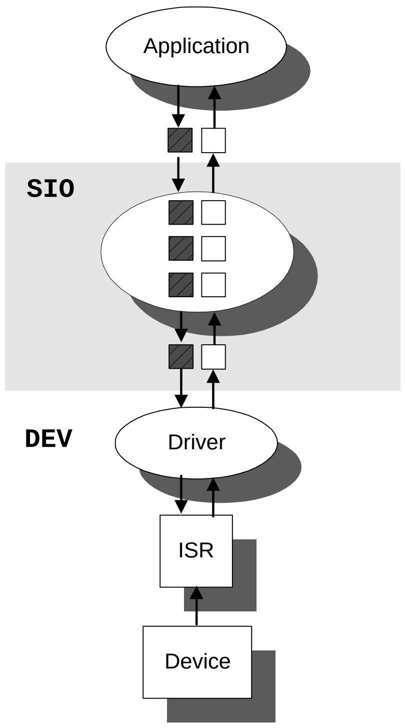
An important aspect of stream I/O is its asynchronous nature. Buffers of data are input or output concurrently with computation. While an application is processing the current buffer, a new input buffer is being filled and a previous one is being output. This efficient management of I/O buffers allows streams to minimize copying of data. Streams exchange pointers rather than data, thus reducing overhead and allowing programs to meet real-time constraints more readily.

A typical program gets a buffer of input data, processes the data, and then outputs a buffer of processed data. This sequence repeats over and over, usually until the program is terminated.

Digital-to-analog converters, video frame grabbers, transducers, and DMA channels are just a few examples of common I/O devices. The stream module (SIO) interacts with these different types of devices through drivers (managed by the DEV module) that use the DSP/BIOS programming interface.

Device drivers are software modules that manage a class of devices. For example, two common classes are serial ports and parallel ports. These modules follow a common interface (provided by DEV) so stream functions can make generic requests, the drivers execute in whatever manner is appropriate for the particular class of devices.

This figure depicts the interaction between streams and devices. The shaded area illustrates the material covered by this chapter: the stream portion of this interaction, handled by the SIO module. Chapter 8 discusses the DEV module and the relationship of streams with devices.



Data pipes are used to buffer streams of input and output data. These data pipes provide a consistent software data structure you can use to drive I/O between the DSP chip and all kinds of real-time peripheral devices. There is more overhead with a data pipe than with streams, and notification is automatically handled by the pipe manager. All I/O operations on a pipe deal with one frame at a time; although each frame has a fixed length, the application may put a variable amount of data in each frame up to the length of the frame.

Separate pipes should be used for each data transfer thread, and a pipe should only have a single reader and a single writer, providing point to point communication. Often one end of a pipe is controlled by a hardware ISR and the other end is controlled by an SWI function. Pipes can also transfer data between two application threads.

Host channel objects allow an application to stream data between the target and the host. Host channels are statically configured for input or output. Each host channel is internally implemented using a data pipe object.

7.2 Comparing Pipes and Streams

DSP/BIOS supports two different models for data transfer. One model is the pipe model, which is used by the PIP and HST modules. The other model is the stream model, which is used by the SIO and DEV modules.

Both models require that a pipe or stream have a single reader thread and a single writer thread. Both models transfer buffers within the pipe or stream by copying pointers rather than by copying data between buffers.

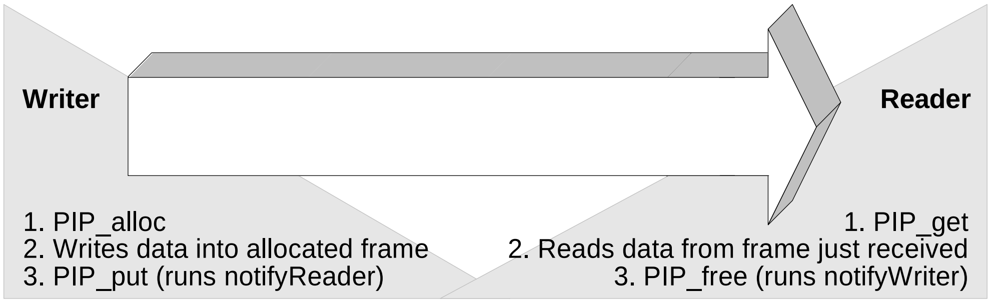
In general, the pipe model supports low-level communication, while the stream model supports high-level, device-independent I/O. The following table compares the two models in more detail.

Pipes (PIP and HST)	**Streams (SIO and DEV)**
Programmer must create own driver structure.	Provides a more structured approach to device-driver creation.
Reader and writer may be any thread type or host PC.	One end must be handled by a task (TSK) using SIO calls. The other end must be handled by an HWI using Dxx calls.
PIP functions are non-blocking. Program must check to make sure a buffer is available before reading from or writing to the pipe.	SIO_put, SIO_get, and SIO_reclaim are blocking functions and will cause task to wait until a buffer is available. (SIO_issue is non-blocking.)
Uses less memory and is generally faster.	More flexible; generally simpler to use.
Each pipe owns its own buffers.	Buffers can be transferred from one stream to another without copying. (In practice, copying is usually necessary anyway because the data is processed.)
Pipes must be created statically with the Configuration Tool.	Streams may be created either at run-time or statically with the Configuration Tool. Streams may be opened by name.
No built-in support for stacking devices.	Support is provided for stacking devices.
Using the HST module with pipes is an easy way to handle data transfer between the host and target.	A number of device drivers are provided with DSP/BIOS.

7.3 Data Pipe Manager (PIP Module)

Pipes are designed to manage block I/O (also called stream-based or asynchronous I/O). Each pipe object maintains a buffer divided into a fixed number of fixed length frames, specified by the numframes and framesize properties. All I/O operations on a pipe deal with one frame at a time. Although each frame has a fixed length, the application may put a variable amount of data in each frame (up to the length of the frame).

A pipe has two ends. The writer end is where the program writes frames of data. The reader end is where the program reads frames of data.


Writer
Reader
1. PIP_alloc
2. Writes data into allocated frame
3. PIP_put (runs notifyReader)
1. PIP_get
2. Reads data from frame just received
3. PIP_free (runs notifyWriter)

Data notification functions (notifyReader and notifyWriter) are performed to synchronize data transfer. These functions are triggered when a frame of data is read or written to notify the program that a frame is free or data is available. These functions are performed in the context of the function that calls PIP_free or PIP_put. They may also be called from the thread that calls PIP_get or PIP_alloc. When PIP_get is called, DSP/BIOS checks whether there are more full frames in the pipe. If so, the notifyReader function is executed. When PIP_alloc is called, DSP/BIOS checks whether there are more empty frames in the pipe. If so, the notifyWriter function is executed.

A pipe should have a single reader and a single writer. Often, one end of a pipe is controlled by a hardware ISR and the other end is controlled by a software interrupt function. Pipes can also be used to transfer data within the program between two application threads.

During program startup (which is described in detail in section 2.5, *DSP/BIOS Startup Sequence*, page 2-17), the BIOS_start function enables the DSP/BIOS modules. As part of this step, the PIP_startup function calls the notifyWriter function for each pipe object, since at startup all pipes have available empty frames.

There are no special format or data type requirements for the data to be transferred by a pipe.

The online help in the Configuration Tool describes data pipe objects and their parameters. See *PIP Module* in the *API Reference Guide* for information on the PIP module API.

7.3.1 Writing Data to a Pipe

The steps that a program should perform to write data to a pipe are as follows:

1) A function should first check the number of empty frames available to be filled with data. To do this, the program must check the return value of PIP_getWriterNumFrames. This function call returns the number of empty frames in a pipe object.
2) If the number of empty frames is greater than 0, the function then calls PIP_alloc to get an empty frame from the pipe.
3) Before returning from the PIP_alloc call, DSP/BIOS checks whether there are additional empty frames available in the pipe. If so, the notifyWriter function is called at this time.
4) Once PIP_alloc returns, the empty frame can be used by the application code to store data. To do this the function needs to know the frame's start address and its size. The API function PIP_getWriterAddr returns the address of the beginning of the allocated frame. The API function PIP_getWriterSize returns the number of words that can be written to the frame. (The default value for an empty frame is the configured frame size.)
5) When the frame is full of data, it can be returned to the pipe. If the number of words written to the frame is less than the frame size, the function can specify this by calling the PIP_setWriterSize function. Afterwards, call PIP_put to return the data to the pipe.
6) Calling PIP_put causes the notifyReader function to run. This enables the writer thread to notify the reader thread that there is data available in the pipe to be read.

The following code fragment demonstrates the previous steps:

```
extern far PIP_Obj writerPipe;   /* pipe object created with
                                    the Configuration Tool */
writer()
{
    Uns size;
    Uns newsize;
    Ptr addr;

    if (PIP_getWriterNumFrames(&writerPipe) > 0) {
        PIP_alloc(&writerPipe);  /* allocate an empty frame */
    }
    else {
        return;      /* There are no available empty frames */
    }

    addr = PIP_getWriterAddr(&writerPipe);
    size = PIP_getWriterSize(&writerPipe);

    ’ fill up the frame ’

    /* optional */
    newsize = ’number of words written to the frame’;
    PIP_setWriterSize(&writerPipe, newsize);

    /* release the full frame back to the pipe */
    PIP_put(&writerPipe);
}
```

7.3.2 Reading Data from a Pipe

To read a full frame from a pipe, a program should perform the following steps:

1) The function should first check the number of full frames available to be read from the pipe. To do this, the program must check the return value of PIP_getReaderNumFrames. This function call returns the number of full frames in a pipe object.

2) If the number of full frames is greater than 0, the function then calls PIP_get to get a full frame from the pipe.

3) Before returning from the PIP_get call, DSP/BIOS checks whether there are additional full frames available in the pipe. If so, the notifyReader function is called at this time.

4) Once PIP_get returns, the data in the full frame can be read by the application. To do this the function needs to know the frame’s start address and its size. The API function PIP_getReaderAddr returns the address of the beginning of the full frame. The API function PIP_getReaderSize returns the number of valid data words in the frame.

5) When the application has finished reading all the data, the frame can be returned to the pipe by calling PIP_free.

6) Calling PIP_free causes the notifyWriter function to run. This enables the reader thread to notify the writer thread that there is a new empty frame available in the pipe.

The following code fragment demonstrates the previous steps:

```
extern far PIP_Obj readerPipe;  /* created with the
                                   Configuration Tool */
reader()
{
    Uns size;
    Ptr addr;

    if (PIP_getReaderNumFrames(&readerPipe) > 0) {
        PIP_get(&readerPipe);    /* get a full frame */
    }
    else {
        return;       /* There are no available full frames */
    }

    addr = PIP_getReaderAddr(&readerPipe);
    size = PIP_getReaderSize(&readerPipe);

    ’ read the data from the frame ’

    /* release the empty frame back to the pipe */
    PIP_free(&readerPipe);
}
```

7.3.3 Using a Pipe’s Notify Functions

The reader or writer threads of a pipe can operate in a polled mode and directly test the number of full or empty frames available before retrieving the next full or empty frame. The example code in section 7.3.1, *Writing Data to a Pipe*, page 7-6, and section 7.3.2, *Reading Data from a Pipe*, page 7-7, demonstrates this type of polled read and write operation.

When used to buffer real-time I/O streams written (read) by a hardware peripheral, pipe objects often serve as a data channel between the HWI routine triggered by the peripheral itself and the program function that ultimately reads (writes) the data. In such situations, the application can effectively synchronize itself with the underlying I/O stream by configuring the pipe’s notifyReader (notifyWriter) function to automatically post a software interrupt that runs the reader (writer) function. When the HWI routine finishes filling up (reading) a frame and calls PIP_put (PIP_free), the pipe’s notify function can be used to automatically post a software interrupt. In this case, rather than polling the pipe for frame availability, the reader (writer) function runs only when the software interrupt is triggered; i.e., when frames are available to be read (written).

Such a function would not need to check for the availability of frames in the pipe, since it is called only when data is ready. As a precaution, the function may still check whether frames are ready, and if not, cause an error condition, as in the following example code:

```
if (PIP_getReaderNumFrames(&readerPipe) = 0) {
   error();  /* reader function should not have been posted! */
}
```

Hence, the notify function of pipe objects can serve as a flow-control mechanism to manage I/O to other threads and hardware devices.

7.3.4 Calling Order for PIP APIs

Each pipe object internally maintains a list of empty frames and a counter with the number of empty frames on the writer side of the pipe, and a list of full frames and a counter with the number of full frames on the reader side of the pipe. The pipe object also contains a descriptor of the current writer frame (i.e., the last frame allocated and currently being filled by the application) and the current reader frame (i.e., the last full frame that the application got and that is currently reading).

When PIP_alloc is called, the writer counter is decreased by 1. An empty frame is removed from the writer list and the writer frame descriptor is updated with the information from this frame. When the application calls PIP_put after filling the frame, the reader counter is increased by one, and the writer frame descriptor is used by DSP/BIOS to add the new full frame to the pipe's reader list.

Note:

Every call to PIP_alloc must be followed by a call to PIP_put before PIP_alloc can be called again: the pipe I/O mechanism does not allow consecutive PIP_alloc calls. Doing so would overwrite previous descriptor information and would produce undetermined results.

```
/* correct */        /* error! */
PIP_alloc();         PIP_alloc();
...                  ...
PIP_put();           PIP_alloc();
...                  ...
PIP_alloc();         PIP_put();
...                  ...
PIP_put();           PIP_put();
```

Similarly when PIP_get is called, the reader counter is decreased by 1. A full frame is removed from the reader list and the reader frame descriptor is updated with the information from this frame. When the application calls PIP_free after reading the frame, the writer counter is increased by 1, and the reader frame descriptor is used by DSP/BIOS to add the new empty frame to the pipe's writer list. Hence, every call to PIP_get must be followed by a call to PIP_free before PIP_get can be called again: the pipe I/O mechanism does not allow consecutive PIP_get calls. Doing so would overwrite previous descriptor information and would produce undetermined results.

```
/* correct */        /* error! */
PIP_get();           PIP_get();
...                  ...
PIP_free();          PIP_get();
...                  ...
PIP_get();           PIP_free();
...                  ...
PIP_free();          PIP_free();
```

7.3.4.1 Avoiding Recursion Problems

Care should be applied when a pipe's notify function calls PIP APIs for the same pipe.

Consider the following example: A pipe's notifyReader function calls PIP_get for the same pipe. The pipe's reader is an HWI routine. The pipe's writer is an SWI routine. Hence the reader has higher priority than the writer. (Calling PIP_get from the notifyReader in this situation may make sense because this allows the application to get the next full buffer ready to be used by the reader—the HWI routine—as soon as it is available and before the hardware interrupt is triggered again.)

The pipe's reader function, the HWI routine, calls PIP_get to read data from the pipe. The pipe's writer function, the SWI routine, calls PIP_put. Since the call to the notifyReader happens within PIP_put in the context of the current routine, a call to PIP_get also happens from the SWI writer routine.

Hence, in the example described two threads with different priorities call PIP_get for the same pipe. This could have catastrophic consequences if one thread preempts the other and as a result, PIP_get is called twice before calling PIP_free, or PIP_get is preempted and called again for the same pipe from a different thread.

Note:

As a general rule to avoid recursion, you should avoid calling PIP functions as part of notifyReader and notifyWriter. If necessary for application efficiency, such calls should be protected to prevent reentrancy for the same pipe object and the wrong calling sequence for the PIP APIs.

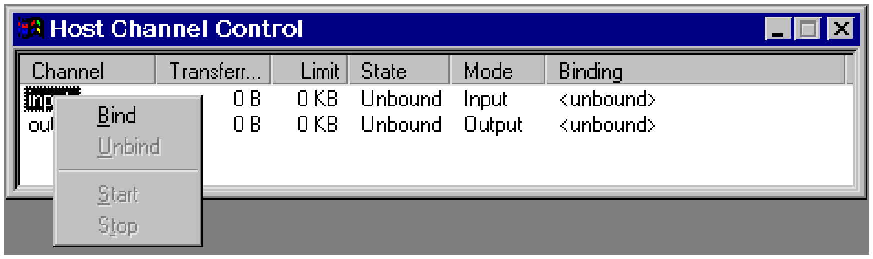
7.4 Host Channel Manager (HST Module)

The HST module manages host channel objects, which allow an application to stream data between the target and the host. Host channels are configured for input or output. Input streams read data from the host to the target. Output streams transfer data from the target to the host.

Note:

HST channel names cannot start with a leading underscore (_).

You dynamically bind channels to files on the PC host by right-clicking on the Host Channel Control in Code Composer. Then you start the data transfer for each channel.


Host Channel Control

Channel	Transferr...	Limit	State	Mode	Binding
inp	0 B	0 KB	Unbound	Input	<unbound>
oul	0 B	0 KB	Unbound	Output	<unbound>

Each host channel is internally implemented using a pipe object. To use a particular host channel, the program uses HST_getpipe to get the corresponding pipe object and then transfers data by calling the PIP_get and PIP_free operations (for input) or PIP_alloc and PIP_put operations (for output).

The code for reading data might look like the following:

```
extern far HST_Obj input;

readFromHost()
{
    PIP_Obj *pipe;
    Uns size;
    Ptr addr;

    pipe = HST_getpipe(&input)    /* get a pointer to the host
                                     channel’s pipe object */
    PIP_get(pipe);                /* get a full frame from the
                                     host */
    size = PIP_getReaderSize(pipe);
    addr = PIP_getReaderAddr(pipe);

    ’ read data from frame ’

    PIP_free(pipe);       /* release empty frame to the host */
}
```

Each host channel can specify a data notification function to be performed when a frame of data for an input channel (or free space for an output channel) is available. This function is triggered when the host writes or reads a frame of data.

HST channels treat files as 32-bit words of raw data. The format of the data is application-specific, and you should verify that the host and the target agree on the data format and ordering. For example, if you are reading 32-bit integers from the host, you need to make sure the host file contains the data in the correct byte order. Other than correct byte order, there are no special format or data type requirements for data to be transferred between the host and the target.

While you are developing a program, you may want to use HST objects to simulate data flow and to test changes made to canned data by program algorithms. During early development, especially when testing signal processing algorithms, the program would explicitly use input channels to access data sets from a file for input for the algorithm and would use output channels to record algorithm output. The data saved to a file with the output host channel can be compared with expected results to detect algorithm errors. Later in the program development cycle, when the algorithm appears sound, you can change the HST objects to PIP objects communicating with other threads or I/O drivers for production hardware.

7.4.1 Transfer of HST Data to the Host

While the amount of usable bandwidth for real-time transfer of data streams to the host ultimately depends on the choice of physical data link, the HST Channel interface remains independent of the physical link. The HST

Manager in the Configuration Tool allows you to choose among the physical connections available.

The actual data transfer to the host occurs during the idle loop, from the LNK_dataPump idle function.

7.5 I/O Performance Issues

If you are using an HST object, the host PC reads or writes data using the function specified by the LNK_dataPump object. This is a built-in IDL object that runs its function as part of the background thread. Since background threads have the lowest priority, software interrupts and hardware interrupts preempt data transfer.

Note that the polling rates you set in the LOG, STS, and TRC controls do not control the data transfer rate for HST objects. (Faster polling rates actually slow the data transfer rate somewhat because LOG, STS, and TRC data also need to be transferred.)

Chapter 8

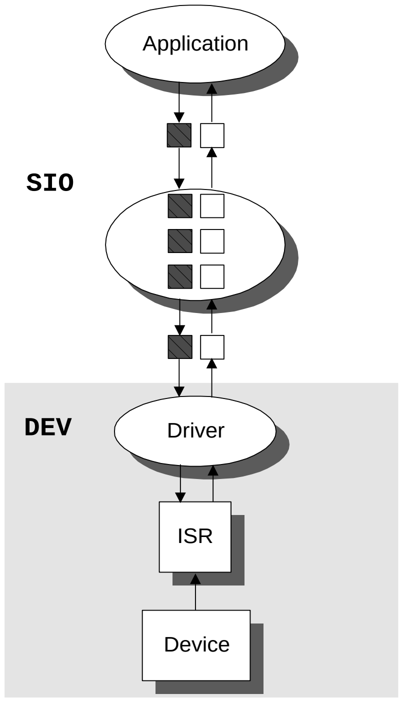
Streaming I/O and Device Drivers

This chapter describes issues relating to writing and using device drivers, and gives several programming examples.

Topic Page

8.1 Overview

Chapter 7 described the device-independent I/O operations supported by DSP/BIOS from the vantage point of an application program. Programs indirectly invoke corresponding functions implemented by the driver managing the particular physical device attached to the stream, using generic functions provided by the SIO module. As depicted in the shaded portion of the figure below, this chapter describes device-independent I/O in DSP/BIOS from the driver's perspective of this interface.



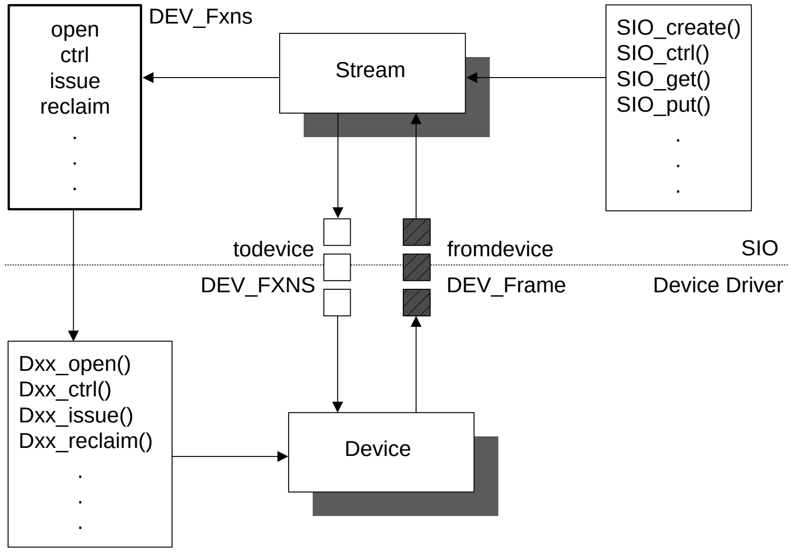
Unlike other modules, your application programs do not issue direct calls to driver functions that manipulate individual device objects managed by the SIO module. Instead, each driver module exports a specifically named structure of a specific type (DEV_Fxns), which in turn is used by the SIO module to route generic function calls to the proper driver function.

As illustrated in the following table, each SIO operation calls the appropriate driver function by referencing this table. Dxx designates the device-specific function which you write for your particular device.

Generic I/O Operation	Internal Driver Operation
SIO_create(name, mode, bufsize, attrs)	Dxx_open(device, name)
SIO_delete(stream)	Dxx_close(device)
SIO_get(stream, &buf)	Dxx_issue(device) and Dxx_reclaim(device)
SIO_put(stream, &buf, nbytes)	Dxx_issue(device and / Dxx_reclaim(device)
SIO_ctrl(stream, cmd, arg)	Dxx_ctrl(device, cmd, arg)
SIO_idle(stream)	Dxx_idle(device, FALSE)
SIO_flush(stream)	Dxx_idle(device, TRUE)
SIO_select(streamtab, n, timeout)	Dxx_ready(device, sem)
SIO_issue(stream, buf, nbytes, arg)	Dxx_issue(device)
SIO_reclaim(stream, &buf, &arg)	Dxx_reclaim(device)
SIO_staticbuf(stream, &buf)	none

As we will see, these internal driver functions can rely on virtually all of the capabilities supplied by DSP/BIOS, ranging from the multitasking features of the kernel to the application-level services. Drivers will even use the device-independent I/O interface of DSP/BIOS to communicate indirectly with other drivers, especially in supporting stackable devices.

The figure below illustrates the relationship between the device, the Dxx device driver, and the stream accepting data from the device. SIO calls the Dxx functions listed in DEV_Fxns, the function table for the device. Both input and output streams exchange buffers with the device using the atomic queues device->todevice and device->fromdevice.



For every device driver you will need to write Dxx_open, Dxx_idle, Dxx_input, Dxx_output, Dxx_close, Dxx_ctrl, Dxx_ready, Dxx_issue, and Dxx_reclaim.

8.2 Creating and Deleting Streams

To use a stream to perform I/O with a device, the device must first be configured in the Configuration Tool. Then, create the stream object in the Configuration Tool or at run-time with the SIO_create function.

8.2.1 Adding a Device with the Configuration Tool

To enable your application to do streaming I/O with a device, the device must first be added and configured with the Configuration Tool. You can add a device for any driver included in the product distribution or a user-supplied driver.

8.2.2 Creating Streams with the Configuration Tool

You can create streams using the Configuration Tool. The Configuration Tool allows you to set the stream attributes for each stream and for the SIO manager itself. You cannot use the SIO_delete function to delete streams created with the Configuration Tool.

Note:

The Configuration Tool cannot be used to create streams for stacking devices, which are described in section 8.4, *Stackable Devices*, page 8-16. You must use SIO_create to create any stream that is used with a stacking device.

8.2.3 Creating and Deleting Streams Dynamically

You can also create a stream at run-time with the SIO_create function.

```
SIO_Handle SIO_create(name, mode, bufsize, attrs)
    String       name;
    Int          mode;
    Uns          bufsize;
    SIO_Attrs    *attrs;
```

SIO_create creates a stream and returns a handle of type SIO_Handle. SIO_create opens the device(s) specified by name, specifying buffers of size bufsize. Optional attributes specify the number of buffers, the buffer memory segment, the streaming model, etc. The mode parameter is used to specify whether the stream is an input (SIO_INPUT) or output (SIO_OUTPUT) stream.

Note:

The name must match the name given to the device in the Configuration Tool preceded by a “/.” For example, for a device called sine, the name should be “/sine.”

If you open the stream with the streaming model (attrs->model) set to SIO_STANDARD (the default), buffers of the specified size are allocated and used to prime the stream. If you open the stream with the streaming model set to SIO_ISSUERECLAIM, no stream buffers are allocated, since the creator of the stream is expected to supply all necessary buffers.

SIO_delete closes the associated device(s) and frees the stream object. If the stream was opened using the SIO_STANDARD streaming model, it also frees all buffers remaining in the stream. User-held stream buffers must be explicitly freed by the user’s code.

```
Int SIO_delete(stream)
    SIO_Handle     stream;
```

8.3 Stream I/O—Reading and Writing Streams

There are two models for streaming data in DSP/BIOS: the standard model and the issue/reclaim model. The standard model provides a simple method for using streams, while the issue/reclaim model provides more control over the stream operation.

SIO_get and SIO_put implement the standard stream model.

SIO_get is used to input the data buffers. SIO_get exchanges buffers with the stream. The bufp parameter is used to pass the device a buffer and return a different buffer to the application. SIO_get returns the number of bytes in the input buffer.

```
Int SIO_get(stream, bufp)
    SIO_Handle      stream;
    Ptr             *bufp;
```

The SIO_put function performs the output of data buffers, and, like SIO_get, exchanges physical buffers with the stream. SIO_put takes the number of bytes in the output buffer.

```
Int SIO_put(stream, bufp, nbytes)
    SIO_Handle    stream;
    Ptr           *bufp;
    Uns           nbytes;
```

Note:

Since the buffer pointed to by bufp is exchanged with the stream, the buffer size, memory segment, and alignment must correspond to the attributes of stream.

SIO_issue and SIO_reclaim are the calls that implement the issue/reclaim streaming model. SIO_issue sends a buffer to a stream. No buffer is returned, and the stream returns control to the task without blocking:

```
Int SIO_issue(stream, pbuf, nbytes, arg)
    SIO_Handle     stream;
    Ptr            pbuf;
    Uns            nbytes;
    Arg            arg;
```

arg is not interpreted by DSP/BIOS, but is offered as a service to the stream client. arg is passed to each device with the associated buffer data. It can be used by the stream client as a method of communicating with the device drivers. For example, arg could be used to send a time stamp to an output device, indicating exactly when the data is to be rendered.

SIO_reclaim requests a stream to return a buffer.

```
Int SIO_reclaim(stream, bufp, parg)
    SIO_Handle     stream;
    Ptr            *bufp;
    Arg            *parg;
```

If no buffer is available, the stream will block the task until the buffer becomes available or the stream's timeout has elapsed.

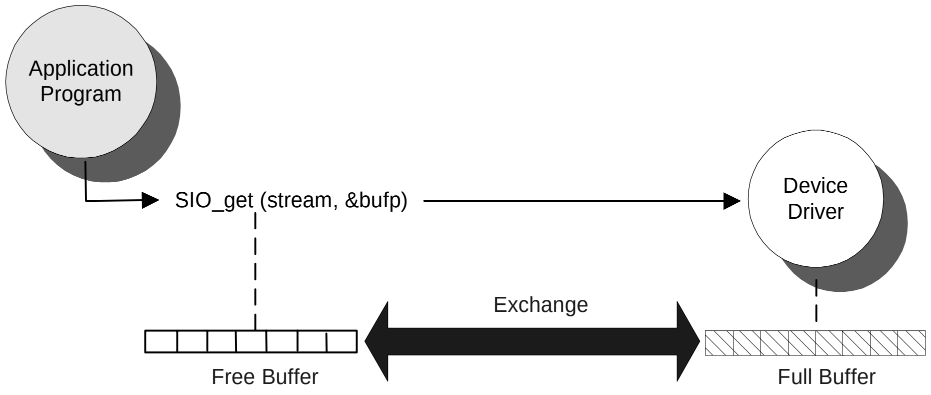
At a basic level, the most obvious difference between the standard and issue/reclaim models is that the issue/reclaim model separates the notification of a buffer's arrival (SIO_issue) and the waiting for a buffer to become available (SIO_reclaim). So, an SIO_issue / SIO_reclaim pair provides the same buffer exchange as calling SIO_get or SIO_put.

The issue/reclaim streaming model provides greater flexibility by allowing the stream client to control the number of outstanding buffers at run-time. A client can send multiple buffers to a stream, without blocking, by using SIO_issue. The buffers are returned, at the client's request, by calling SIO_reclaim. This allows the client to choose how deep to buffer a device and when to block and wait for a buffer.

The issue/reclaim streaming model also provides greater determinism in buffer management by guaranteeing that the client's buffers is returned in the order that they were issued. This allows a client to use memory from any source for streaming. For example, if a DSP/BIOS task receives a large buffer, that task can pass the buffer to the stream in small pieces—simply by advancing a pointer through the larger buffer and calling SIO_issue for each piece. This will work because each piece of the buffer is guaranteed to come back in the same order it was sent.

8.3.1 Buffer Exchange

An important part of the streaming model in DSP/BIOS is buffer exchange. To provide efficient I/O operations with a low amount of overhead, DSP/BIOS avoids copying data from one place to another during certain I/O operations. Instead, DSP/BIOS uses SIO_get, SIO_put, SIO_issue, and SIO_reclaim to move buffer pointers to and from the device. The figure below shows a conceptual view of how SIO_get works:



In this figure, the device driver associated with stream fills a buffer as data becomes available. At the same time, the application program is processing the current buffer. When the application uses SIO_get to get the next buffer, the new buffer that was filled by the input device is swapped for the buffer passed in. This is accomplished by exchanging buffer pointers instead of copying bufsize bytes of data, which would be very time consuming. Therefore, the overhead of SIO_get is independent of the buffer size.

Note that in each case, the actual physical buffer has been changed by SIO_get. The important implication is that you must make sure that any references to the buffer used in I/O are updated after each operation. Otherwise, you are referencing an invalid buffer.

SIO_put uses the same exchange of pointers to swap buffers for an output stream. SIO_issue and SIO_reclaim each move data in only one direction. Therefore, an SIO_issue / SIO_reclaim pair result in the same swapping of buffer pointers.

Note:

A single stream cannot be used by more than one task simultaneously. That is, only a single task can call SIO_get / SIO_put or SIO_issue / SIO_reclaim at once for each stream in your application.

8.3.2 Example - Reading Input Buffers from a DGN Device

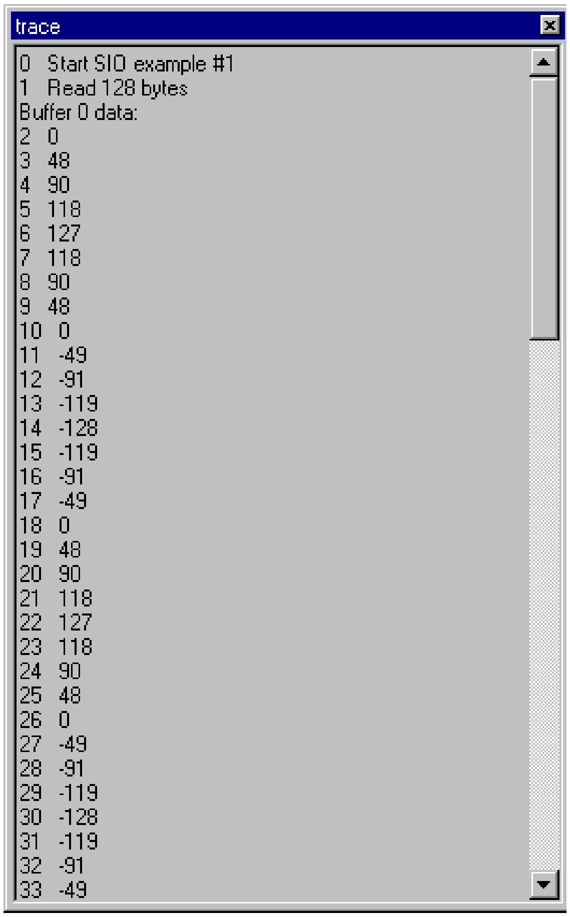
The following example program, in c:\mysrc62\siotest\siotest1.c, illustrates some of the basic SIO functions and provides a straightforward example of reading from a stream. For a complete description of the DGN software generator driver, see the DGN section the *API Reference Guide*.

The configuration template for this example can be found in the siotest directory of the DSP/BIOS distribution. A DGN device called sineWave is used as a data generator to the SIO stream inputStream. The task streamTask calls the function doStreaming to read the sine data from the inputStream and prints it to the log buffer trace.

```
/*
 *  ======== siotest1.c ========
 *  In this program a task reads data from a DGN sine device
 *  and prints the contents of the data buffers to a log buffer.
 *  The data exchange between the task and the device is done
 *  in a device independent fashion using the SIO module APIs.
 *
 *  The stream in this example follows the SIO_STANDARD streaming
 *  model and is created using the Configuration Tool.
 *
 */

#include <std.h>

#include <log.h>
#include <sio.h>
#include <sys.h>
#include <tsk.h>

extern Int IDRAM1;     /* MEM segment ID defined by Conf tool */
extern LOG_Obj trace;  /* LOG object created with Conf tool */
extern SIO_Obj inputStream; /* SIO object created w Conf tool */
extern TSK_Obj streamTask;  /* pre-created task */

SIO_Handle input = &inputStream; /* SIO handle used below */

Void doStreaming(Uns nloops);    /* function for streamTask */

/*
 *  ======== main ========
 */
Void main()
{
    LOG_printf(&trace, "Start SIO example #1");
}
```

```
/*
 *  ======== doStreaming ========
 *  This function is the body of the pre-created TSK thread
 *  streamTask.
 */
Void doStreaming(Uns nloops)
{
    Int i, j, nbytes;
    Int *buf;
    status = SIO_staticbuf(input, (Ptr *)&buf);
    if (status ! = SYS_ok) {
           SYS_abort(“could not acquire static frame:);
    }

    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input, (Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer %d", i);
        }

        LOG_printf(&trace, "Read %d bytes\nBuffer %d data:",
nbytes, i);
        for (j = 0; j < nbytes / sizeof(Int); j++) {
            LOG_printf(&trace, "%d", buf[j]);
        }
    }

    LOG_printf(&trace, "End SIO example #1");
}
```

The output for this example appears as sine wave data in the traceLog window.



```
0   Start SIO example #1
1   Read 128 bytes
Buffer 0 data:
2   0
3   48
4   90
5   118
6   127
7   118
8   90
9   48
10  0
11  -49
12  -91
13  -119
14  -128
15  -119
16  -91
17  -49
18  0
19  48
20  90
21  118
22  127
23  118
24  90
25  48
26  0
27  -49
28  -91
29  -119
30  -128
31  -119
32  -91
33  -49
```

8.3.3 Example - Reading and Writing to a DGN Device

This example adds a new SIO operation to the previous one. An output stream, outputStream, has been added with the Configuration Tool. streamTask reads buffers from a DGN sine device as before, but now it sends the data buffers to outputStream rather than printing the contents to a log buffer:

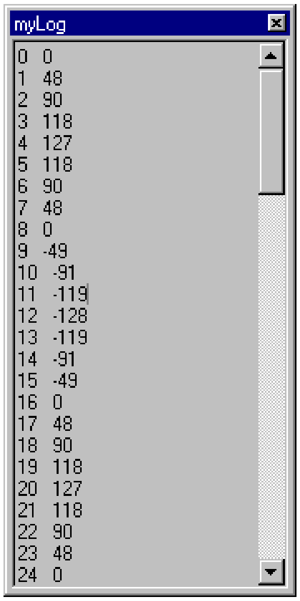
```
======== Portion of siotest2.c ========
/* SIO objects created with conf tool  */
extern SIO_Obj  inputStream;
extern SIO_Obj  outputStream;
SIO_Handle input = &inputStream;
SIO_Handle output = &outputStream;
...

Void doStreaming(Uns nloops)
{
    Int i, j, nbytes;
    Int *buf;
     status = SIO_staticbuf(input, (Ptr *)&buf);
     if (status ! = SYS_ok) {
            SYS_abort(“could not acquire static frame:);
     }

    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input, (Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer %d", i);
        }

        LOG_printf(&trace, "Read %d bytes\nBuffer %d data:",
nbytes, i);
        for (j = 0; j < nbytes / sizeof(Int); j++) {
            LOG_printf(&trace, "%d", buf[j]);
        }
    }

    LOG_printf(&traceLog, "End SIO example #2");
}
```

outputStream sends the data to a DGN user device called printData. printData takes the data buffers received and uses the DGN_print2log function to display their contents in a log buffer. The log buffer is specified by the user in the Configuration Tool.

```
/*  ======== DGN_print2log ========
 *  User function for the DGN user device printData. It takes
as an argument
 *  the address of the LOG object where the data stream should
be printed. */

Void DGN_print2log(Arg arg, Ptr addr, Uns nbytes)
{
    Int      i;
    Int     *buf;
    buf = (Int *)addr;

    for (i = 0; i < nbytes/sizeof(Int); i++) {
LOG_printf((LOG_Obj *)arg, "%d", buf[i]);
    }
}
```

The complete source code and configuration template for this example can be found in the C:\ti\c6000\examples\bios\siotest directory of the DSP/BIOS product distribution (siotest2.c, siotest2.cdb, dgn_print.c). For more details on how to add and configure a DGN device using the Configuration Tool, see the DGN in the *API Reference Guide*.

In the output for this example, sine wave data appears in the myLog window display.



```
0  0
1  48
2  90
3  118
4  127
5  118
6  90
7  48
8  0
9  -49
10  -91
11  -119
12  -128
13  -119
14  -91
15  -49
16  0
17  48
18  90
19  118
20  127
21  118
22  90
23  48
24  0
```

8.3.4 Example - Stream I/O using the Issue/Reclaim Model

The following example is functionally equivalent to the previous one. However, the streams are now created using the Issue/Reclaim model, and the SIO operations to read and write data to a stream are SIO_issue and SIO_reclaim.

In this model, when streams are created dynamically, no buffers are initially allocated so that the application will have to allocate the necessary buffers and provide them to the streams to be used for data I/O. For static streams, you can allocate static buffers with the Configuration Tool by checking the "Allocate Static Buffer(s)" check box for the SIO object.

```
/*  ======== doIRstreaming ======== */
Void doIRstreaming(Uns nloops)
{
    Ptr    buf;
    Arg    arg;
    Int    i, nbytes;

    /* Prime the stream with a couple of buffers */
    buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
    if (buf == MEM_ILLEGAL) {
        SYS_abort("Memory allocation error");
    }
    /* Issue an empty buffer to the input stream */
    if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0) {
        SYS_abort("Error issuing buffer %d", i);
    }

    buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
    if (buf == MEM_ILLEGAL) {
        SYS_abort("Memory allocation error");
    }

    for (i = 0; i < nloops; i++) {
        /* Issue an empty buffer to the input stream */
      if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0) {
            SYS_abort("Error issuing buffer %d", i);
        }
        /* Reclaim full buffer from the input stream */
        if ((nbytes = SIO_reclaim(input, &buf, &arg)) < 0) {
            SYS_abort("Error reclaiming buffer %d", i);
        }
        /* Issue full buffer to the output stream */
        if (SIO_issue(output, buf, nbytes, NULL) < 0) {
            SYS_abort("Error issuing buffer %d", i);
        }
        /* Reclaim empty buffer from the output stream to be
reused */
        if (SIO_reclaim(output, &buf, &arg) < 0) {
            SYS_abort("Error reclaiming buffer %d", i);
        }
    }
    /* Reclaim and delete the buffers used */
    MEM_free(IDRAM1, buf, SIO_bufsize(input));
    if ((nbytes = SIO_reclaim(input, &buf, &arg)) < 0) {
        SYS_abort("Error reclaiming buffer %d", i);
    }
    if (SIO_issue(output, buf, nbytes, NULL) < 0) {
        SYS_abort("Error issuing buffer %d", i);
        }
    if (SIO_reclaim(output, &buf, &arg) < 0) {
        SYS_abort("Error reclaiming buffer %d", i);
    }

    MEM_free(IDRAM1, buf, SIO_bufsize(input));
}
```

The complete source code and configuration template for this example can be found in the C:\ti\c6000\examples\bios\siotest directory of the DSP/BIOS product distribution (siotest3.c, siotest3.cdb, dgn_print.c).

The output for this example is the same as for siotest2.

8.4 Stackable Devices

The capabilities of the SIO module play an important role in fostering device-independence within DSP/BIOS in that logical devices insulate your application programs from the details of designating a particular device. For example, /dac is a logical device name that does not imply any particular DAC hardware. The device-naming convention adds another dimension to device-independent I/O which is unique to DSP/BIOS—the ability to use a single name to denote a stack of devices.

Note:

By stacking certain data streaming or message passing devices atop one another, you can create “virtual” I/O devices that further insulate your applications from the underlying system hardware.

Consider, as an example, a program implementing an algorithm that inputs and outputs a stream of fixed-point data using a pair of A/D-D/A converters. However, the A/D-D/A device can take only the 14 most significant bits of data, and the other 2 bits have to be zero if you want to scale up the input data.

Instead of cluttering the program with excess code for data conversion and buffering to satisfy the algorithm’s needs, we can open a pair of “virtual” devices that implicitly perform a series of transformations on the data produced and consumed by the underlying real devices.

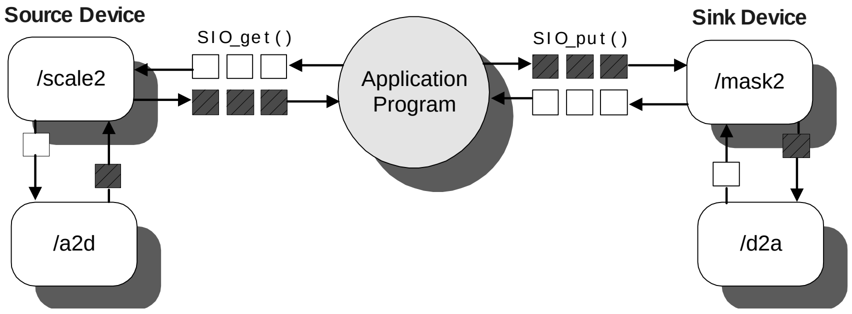
```
SIO_Handle input;
SIO_Handle output;
Ptr        buf;
Int        n;

buf = MEM_alloc(0, MAXSIZE, 0);

input = SIO_create("/scale2/a2d", SIO_INPUT, MAXSIZE, NULL);
output = SIO_create("/mask2/d2a", SIO_OUTPUT, MAXSIZE, NULL);

while (n = SIO_get(input, &buf)) {

    ‘apply algorithm to contents of buf‘

    SIO_put(output, &buf, n);
}

SIO_delete(input);
SIO_delete(output);
```

The virtual input device, /scale2/a2d, actually comprises a stack of two devices, each named according to the prefix of the device name specified in your configuration file.

1) /scale2 designates a device that transforms a fixed-point data stream produced by an underlying device (/a2d) into a stream of scaled fixed-point values; and

2) /a2d designates a device managed by the A/D-D/A device driver that produces a stream of fixed-point input from an A/D converter.

The virtual output device, /mask2/d2a, likewise denotes a stack of two devices. This figure shows the flow of empty and full frames through these virtual source and sink devices as the application program calls the SIO data streaming functions:



8.4.1 Example - SIO_create and Stacking Devices.

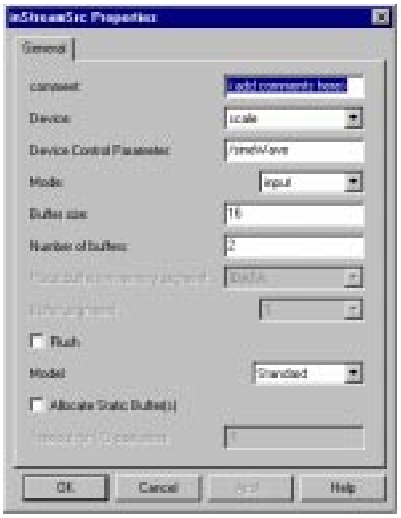
In the following example two tasks, sourceTask and sinkTask, exchange data through a pipe device.

sourceTask is a writer task that receives data from an input stream attached to a DGN sine device and redirects the data to an output stream attached to a DPI pipe device. The input stream has also a stacking device, scale, on top of the DGN sine device. The data stream coming from sine is first processed by the scale device (that multiplies each data point by a constant integer value), before it is received by sourceTask.

sinkTask is a reader task that reads the data that sourceTask sent to the DPI pipe device through an input stream, and redirects it to a DGN printData device through an output stream.

The devices in this example have been configured with the Configuration Tool. The complete source code and configuration template for this example can be found in the C:\ti\c6000\examples\bios\siotest directory of the DSP/BIOS product distribution (siotest5.c, siotest5.cdb, dgn_print.c). The devices sineWave and printDat are DGN devices. pip0 is a DPI device. scale is a DTR stacking device. For more information on how to add and configure DPI, DGN, and DTR devices, see the DPI, DGN and DTR drivers description in the *API Reference Guide*.

The streams in this example have also been added using the Configuration Tool. The input stream for the sourceTask task is inStreamSrc and has been configured as follows:



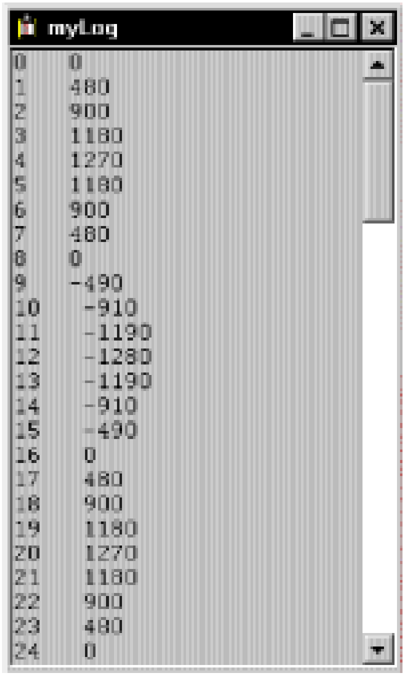
comment: <add comments here>
Device: scale
Device Control Parameter: /sineWave
Mode: input
Buffer size: 16
Number of buffers: 2
Place buffers in memory segment: DATA
Buffer alignment: 1
☐ Flush
Model: Standard
☐ Allocate Static Buffer(s)
Timeout for I/O operation: 1

When you add an SIO stream in the Configuration Tool that uses a stacking device, you must first enter a configured terminal device in the Device Control Parameter property box. The name of the terminal device must be preceded by "/". In the example we use "/sineWave", where sineWave is the name of a configured DGN terminal device. Then select the stacking device (scale) from the dropdown list in the Device property. The Configuration Tool will not allow you to select a stacking device in Device until a terminal device has been entered in Device Control Parameter. The other SIO streams created for the example are outStreamSrc (output stream for sourceTask), inStreamSink (input stream for sinkTask), and outStreamSink (output stream for sinkTask). The devices used by these streams are the terminal devices pip0 and printData.

```
/*
 *  ======== siotest5.c ========
 *  In this program two tasks are created that exchange data
 *  through a pipe device. The source task reads sine wave data
 *  from a DGN device through a DTR device stacked on the sine
 *  device, and then writes it to a pipe device. The sink task
 *  reads the data from the pipe device and writes it to the
 *  printData DGN device. The data exchange between the tasks
 *  and the devices is done in a device independent fashion
 * using the SIO module APIs.
 *
 *  The streams in this example follow the SIO_STANDARD streaming
 *  model and are created with the Configuration Tool.
 */

#include <std.h>

#include <dtr.h>
#include <log.h>
#include <mem.h>
#include <sio.h>
#include <sys.h>
#include <tsk.h>

#define BUFSIZE 128

#ifdef _62_
#define SegId IDRAM
extern Int IDRAM;/* MEM segment ID defined with conf tool */
#endif

#ifdef _54_
#define SegId IDATA
extern Int IDATA;/* MEM segment ID defined with conf tool */
#endif

#ifdef _55_
#define SegId DATA
extern Int DATA;/* MEM segment ID defined with conf tool */
#endif

extern LOG_Obj trace;/* LOG object created with conf tool */
extern TSK_Obj sourceTask;/* TSK thread objects created via
conf tool */
extern TSK_Obj sinkTask;
extern SIO_Obj inStreamSrc;/* SIO streams created via conf tool
*/
extern SIO_Obj outStreamSrc;
extern SIO_Obj inStreamSink;
extern SIO_Obj outStreamSink;

/* Parameters for the stacking device  "/scale" */
DTR_Params DTR_PRMS = {
    20,   /* Scaling factor */
    NULL,
    NULL
};

Void source(Uns nloops);/* function body for sourceTask above
*/
Void sink(Uns nloops);/* function body for sinkTask above */

static Void doStreaming(SIO_Handle input, SIO_Handle output,
Uns nloops);

/*
 *  ======== main ========
 */
Void main()
{
    LOG_printf(&trace, "Start SIO example #5");
}

/*
 *  ======== source ========
 *  This function forms the body of the sourceTask TSK thread.
 */
Void source(Uns nloops)
{
    SIO_Handle input = &inStreamSrc;
    SIO_Handle output = &outStreamSrc;

    /* Do I/O */
    doStreaming(input, output, nloops);
}

/*
 *  ======== sink ========
 *  This function forms the body of the sinkTask TSK thread.
 */
Void sink(Uns nloops)
{
    SIO_Handle input = &inStreamSink;
    SIO_Handle output = &outStreamSink;

    /* Do I/O */
    doStreaming(input, output, nloops);

    LOG_printf(&trace, "End SIO example #5");
}

/*
 *  ======== doStreaming ========
 *  I/O function for the sink and source tasks.
 */
static Void doStreaming(SIO_Handle input, SIO_Handle output,
Uns nloops)
{
    Ptr    buf;
    Int    i, nbytes;

    if (SIO_staticbuf(input, &buf) == 0){
         SYS_abort("Eror reading buffer %d", i);
     }
     for (i = 0; i < nloops; i++) {
          if ((nbytes = SIO_get (input, &buf)) <0) {
                SYS_abort ("Error reading buffer %d", i);
          }
```

```
            if (SIO_put (output, &buf, nbytes) <0) {
                    SYS_abort ("Error writing buffer %d", i);
            }
        }
}
```

In the output for this example, scaled sine wave data appears in the myLog window display.



```
0    0
1    480
2    900
3    1180
4    1270
5    1180
6    900
7    480
8    0
9    -490
10    -910
11    -1190
12    -1280
13    -1190
14    -910
15    -490
16    0
17    480
18    900
19    1180
20    1270
21    1180
22    900
23    480
24    0
```

You can edit sioTest5.c and change the scaling factor of the DTR_PRMS, rebuild the executable and see the differences in the output to myLog.

A version of this example, where the streams are created dynamically at runtime by calling SIO_create is available in the product distribution (siotest4.c, siotest4.cdb).

8.5 Controlling Streams

A physical device typically requires one or more specialized control signals in order to operate as desired. SIO_ctrl makes it possible to communicate with the device, passing it commands and arguments. Since each device will admit only specialized commands, you will need to consult the documentation for each particular device. The general calling format is shown below:

```
Int SIO_ctrl(stream, cmd, arg)
    SIO_Handle stream;
    Uns        cmd;
    Ptr        arg;
```

The device associated with stream is passed the command represented by the device-specific cmd. A generic pointer to the command's arguments is also passed to the device. The actual control function that is part of the device driver then interprets the command and arguments and acts accordingly.

Assume that an analog-to-digital converter device /a2d has a control operation to change the sample rate. The sample rate might be changed to 12 kHz as follows:

```
SIO_Handle     stream;

stream = SIO_create("/a2d", ...);

SIO_ctrl(stream, DAC_RATE, 12000);
```

In some situations, you may need to synchronize with an I/O device that is doing buffered I/O. There are two methods to synchronize with the devices: SIO_idle and SIO_flush. Either function will leave the device in the idled state. Idling a device means that all buffers are returned to the queues that they were in when the device was initially created. That is, the device is returned to its initial state, and streaming is stopped.

For an input stream, the two functions have the same results: all unread input is lost. For an output stream, SIO_idle will block until all buffered data has been written to the device. However, SIO_flush will simply discard any data that has not already been written to the device. SIO_flush does not block.

The calling sequences for SIO_idle and SIO_flush are shown next:

```
Void SIO_idle(stream);
    SIO_Handle     stream;
Void SIO_flush(stream);
    SIO_Handle     stream;
```

An idle stream does not perform I/O with its underlying device. Thus, a stream can be "turned off" until further input or output is needed by calling SIO_idle or SIO_flush.

8.6 Selecting Among Multiple Streams

The SIO_select function allows a single DSP/BIOS task to wait until an I/O operation may be performed on one or more of a set of SIO streams without blocking. For example, this mechanism is useful in the following applications:

- ❏ Non-blocking I/O. Real-time tasks that stream data to a slow device (e.g., a disk file) must insure that SIO_put does not block.
- ❏ Multitasking. In virtually any multitasking application there are daemon tasks that route data from several sources. The SIO_select mechanism allows a single task to handle all of these sources.

SIO_select is called with an array of streams, an array length, and a time-out value. SIO_select will block (if timeout is not 0) until one of the streams is ready for I/O or the time-out expires. In either case, the mask returned by SIO_select indicates which devices are ready for service (a 1 in bit j indicates that streamtab[j] is ready).

```
Uns SIO_select(streamtab, nstreams, timeout)
    SIO_Handle     streamtab[];   /* stream table */
    Uns            nstreams;      /* number of streams */
    Uns            timeout;       /* return after this many */
                                  /* system clock ticks */
```

8.6.1 Programming Example

In the following example two streams are polled to see if an I/O operation will block:

```
SIO_Handle     stream0;
SIO_Handle     stream1;
SIO_Handle     streamtab[2];
Uns            mask;

...

streamtab[0] = stream0;
streamtab[1] = stream1;

while ((mask = SIO_select(streamtab, 2, 0)) == 0) {

    ‘I/O would block, do something else‘

}

if (mask & 0x1) {
    ‘service stream0‘
}
if (mask & 0x2) {
    ‘service stream1‘
}
```

8.7 Streaming Data to Multiple Clients

A common problem in multiprocessing systems is the simultaneous transmission of a single data buffer to multiple tasks in the system. Such multi-cast transmission, or scattering of data, can be done easily with DSP/BIOS SIO streams. Consider the situation in which a single processor sends data to four client processors.

Streaming data between processors in this context is somewhat different from streaming data to or from an acquisition device, such as an A/D converter, in that a single buffer of data must go to one or more clients. The DSP/BIOS SIO functions SIO_get / SIO_put are used for data I/O. Consider the following function call:

```
SIO_put(inStream, (Ptr)&bufA, npoints);
```

SIO_put will automatically perform a buffer exchange between the buffer already at the device level and the application buffer, and as a result the user no longer has control over the buffer since it is enqueued for I/O, and this I/O happens asynchronously at the interrupt level. This forces the user to copy data in order to send it to multiple clients:

```
‘fill bufA with data‘
for (‘all data points‘) {
    bufB[i] = bufC[i] = bufD[i] ... = bufA[i];
}
SIO_put(outStreamA, (Ptr)&bufA, npoints);
SIO_put(outStreamB, (Ptr)&bufB, npoints);
SIO_put(outStreamC, (Ptr)&bufC, npoints);
SIO_put(outStreamD, (Ptr)&bufD, npoints);
```

Copying the data wastes CPU cycles and requires more memory, since each stream needs buffers. If you were double-buffering, the above example would require eight buffers (two for each stream).

The following example illustrates the advantage of SIO_issue and SIO_reclaim in this situation. The application performs no copying, and it uses only two buffers:

```
SIO_issue(outStreamA, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamB, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamC, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamD, (Ptr)bufA, npoints, NULL);
```

In each call, SIO_issue simply enqueues the buffer pointed to by bufA onto outStream’s todevice queue without blocking. Since there is no copying or blocking, this method greatly reduces the time between having a buffer of data ready for transmission and the time the buffer can be sent to all clients.

Note:

Using SIO_issue to send the same buffer to multiple devices will not work with device drivers that modify the data in the buffer, since the buffer is simultaneously being sent to multiple devices. For example, a stacking device that transforms packed data to unpacked data is modifying the buffer at the same time that another device is outputting the buffer.

In order to remove the buffers from the output devices, corresponding SIO_reclaim functions must be called:

```
SIO_reclaim(outStreamA, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamB, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamC, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamD, (Ptr)&bufA, NULL, SYS_FOREVER);
```

The SIO_issue interface provides a method for allowing all communications drivers access to the same buffer of data. Each communications device driver, which typically will use DMA transfers, will then be transferring this buffer of data concurrently. You will not return from the four SIO_reclaims until a buffer is available from all of the streams.

In summary, the SIO_issue / SIO_reclaim functions offer the most efficient method for the simultaneous transmission of data to more than one stream. Note that this is not a reciprocal operation: the SIO_issue / SIO_reclaim model solves the scatter problem quite efficiently for output, but will not accommodate gathering multiple data sources into a single buffer for input.

8.8 Streaming Data Between Target and Host

Using the Configuration Tool, you can create “host channel objects” (FIO objects), which allow an application to stream data between the target and files on the host. In DSP/BIOS plug-ins, you bind these channels to host files and start them.

DSP/BIOS includes a host I/O module (FIO) that makes it easy to transfer data between the host computer and target program. Each host channel is internally implemented using an SIO stream object. To use a host channel, the program calls FIO_getstream to get the corresponding stream handle, and then transfers the data using SIO calls on the stream.

You configure host channels, or FIO objects, for input or output using the Configuration Tool. Input channels transfer data from the host to the target, and output channels transfer data from the target to the host.

8.9 Configuring the Device Driver

For details about configuring device drivers, including both custom drivers and the drivers provided with DSP/BIOS, you need to reference the specific device driver.

Since device drivers interact directly with hardware, the low-level details of device drivers may vary considerably. However, all device drivers must present the same interface to SIO. In the following sections, an example driver template called Dxx is presented. The template contains (mainly) C code for higher-level operations and pseudocode for lower-level operations. Any device driver should adhere to the standard behavior indicated for the Dxx functions.

You should study the Dxx driver template along with one or more actual drivers. You can also refer to the Dxx functions in the *API Reference Guide* where xx denotes any two-letter combination.

8.9.1 Typical File Organization

Device drivers are usually split into multiple files. For example:

❏ dxx.h—Dxx header file

❏ dxx.c—Dxx functions

❏ dxx_asm.s##—(optional) assembly language functions

Most of the device driver code can be written in C. The following description of Dxx does not use assembly language. However, interrupt service routines are usually written in assembly language for efficiency, and some hardware control functions may also need to be written in assembly language.

We recommend that you become familiar at this point with the layout of one of the software device drivers, such as DGN. In particular, you should note the following points:

1) The header file, dxx.h, will typically contain the following required statements, in addition to any device-specific definitions:

```
/*
 *  ======== dxx.h ========
 */

#include <dev.h>
extern DEV_Fxns    Dxx_FXNS;
/*
 * ======== Dxx_Params ========
 */
typedef struct {

    ‘device parameters go here‘

} Dxx_Params;
```

2) Device parameters, such as Dxx_Params, are specified as properties of the device object in the Configuration Tool.

3) The required table of device functions is contained in dxx.c.

```
DEV_Fxns Dxx_FXNS = {
    Dxx_close,
    Dxx_ctrl,
    Dxx_idle,
    Dxx_issue
    Dxx_open,
    Dxx_ready,
    Dxx_reclaim
};
```

This table is used by the SIO module to call specific device driver functions. For example, SIO_put uses this table to find and call Dxx_issue/Dxx_reclaim.

8.10 DEV Structures

The DEV_Fxns structure contains pointers to internal driver functions corresponding to generic I/O operations.

```
typedef struct DEV_Fxns {
    Int     (*close)(DEV_Handle);
    Int     (*ctrl)(DEV_Handle, Uns, Arg);
    Int     (*idle)(DEV_Handle, Bool);
    Int     (*issue(DEV_Handle);
    Int     (*open)(DEV_Handle, String);
    Bool    (*ready)(DEV_Handle, SEM_Handle);
    Int     (*reclaim)(DEV_Handle);
} DEV_Fxns;
```

Device frames are structures of type DEV_Frame used by SIO and device drivers to enqueue/dequeue stream buffers. The device->todevice and device->fromdevice queues contain elements of this type:

```
typedef struct DEV_Frame {
    QUE_Elem   link;
    Ptr        addr;
    Uns        size;
    Arg        misc;
    Arg        arg;
} DEV_Frame;
```

❑ link is used by QUE_put and QUE_get to enqueue/dequeue the frame.

❑ addr contains the address of the stream buffer.

❑ size contains the logical size of the stream buffer. size may be less than the physical buffer size.

❑ misc is an extra field which is reserved for use by a device.

❑ arg is an extra field available for you to associate information with a particular frame of data. This field should be preserved by the device.

Device driver functions take a DEV_Handle as their first or only parameter, followed by any additional parameters. The DEV_Handle is a pointer to a DEV_Obj, which is created and initialized by SIO_create and passed to Dxx_open for additional initialization. Among other things, a DEV_Obj contains pointers to the buffer queues which SIO and the device use to exchange buffers. All driver functions take a DEV_Handle as their first parameter.

```
typedef DEV_Obj *DEV_Handle;
typedef struct DEV_Obj {
    QUE_Handle todevice;
    QUE_Handle fromdevice;
    Uns        bufsize;
    Uns        nbufs;
    Int        segid;
    Int        mode;
    Int        devid;
    Ptr        params;
    Ptr        object;
    DEV_Fxns   fxns;
    Uns        timeout;
   } DEV_Obj;
```

- todevice is used to transfer DEV_Frame frames to the device. In the SIO_STANDARD (DEV_STANDARD) streaming model, SIO_put puts full frames on this queue, and SIO_get puts empty frames here. In the SIO_ISSUERECLAIM (DEV_ISSUERECLAIM) streaming model, SIO_issue places frames on this queue.
- **fromdevice** is used to transfer DEV_Frame frames from the device. In the SIO_STANDARD (DEV_STANDARD) streaming model, SIO_put gets empty frames from this queue, and SIO_get gets full frames from here. In the SIO_ISSUERECLAIM (DEV_ISSUERECLAIM) streaming model, SIO_reclaim retrieves frames from this queue.
- bufsize specifies the physical size of the buffers in the device queues.
- nbufs specifies the number of buffers allocated for this device in the SIO_STANDARD streaming model, or the maximum number of outstanding buffers in the SIO_ISSUERECLAIM streaming model.
- segid specifies the segment from which device buffers were allocated (SIO_STANDARD).
- mode specifies whether the device is an input (DEV_INPUT) or output (DEV_OUTPUT) device.
- devid is the device ID.
- params is a generic pointer to any device-specific parameters. Some devices have additional parameters which are found here.
- object is a pointer to the device object. Most devices create an object that is referenced in successive device operations.
- fxns is a DEV_Fxns structure containing the driver's functions. This structure is usually a copy of Dxx_FXNS, but it is possible for a driver to dynamically alter these functions in Dxx_open.
- timeout specifies the number of system ticks that SIO_reclaim will wait for I/O to complete.

Only the object and fxns fields should ever be modified by a driver's functions. These fields are essentially output parameters of Dxx_open.

8.11 Device Driver Initialization

The driver function table Dxx_FXNS is initialized in dxx.c, as shown above.

Additional initialization is performed by Dxx_init. The Dxx module is initialized when other application-level modules are initialized. Dxx_init typically calls hardware initialization routines and initializes static driver structures.

```
/*
 *  ======== Dxx_init ========
 */

Void Dxx_init()
{
    ‘Perform hardware initialization‘
}
```

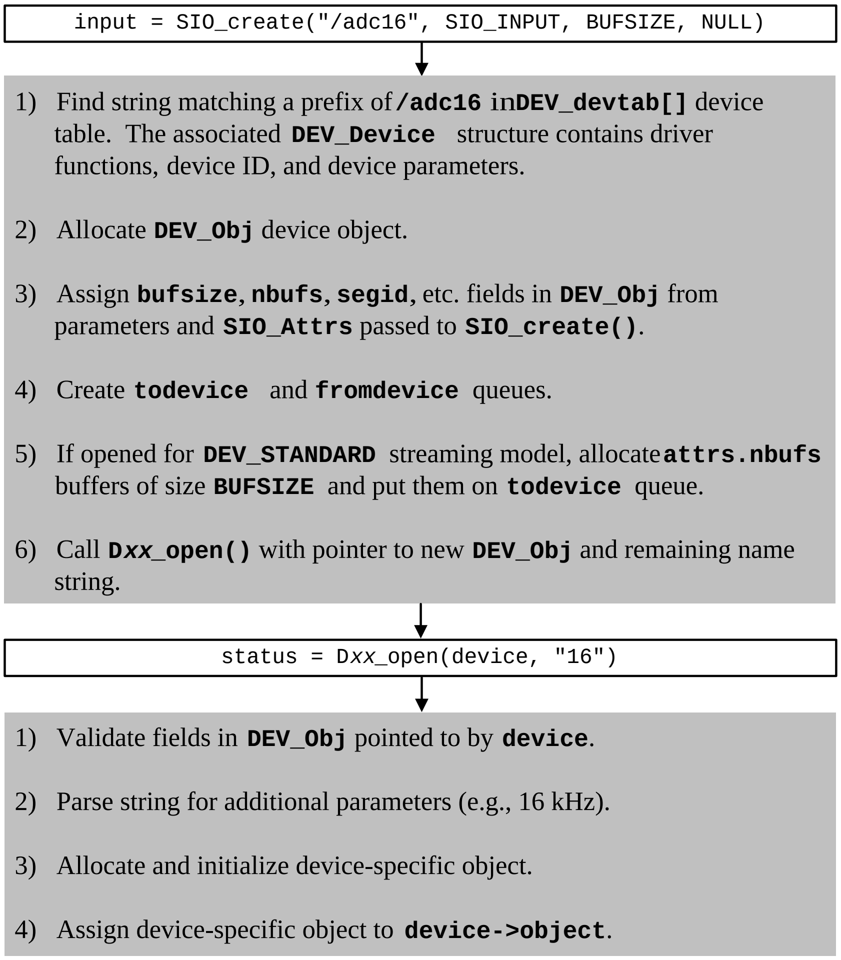
Although Dxx_init is required in order to maintain consistency with DSP/BIOS configuration and initialization standards, there are actually no DSP/BIOS requirements for the internal operation of Dxx_init. There is in fact no standard for hardware initialization, and it may be more appropriate on some systems to perform certain hardware setup operations elsewhere in Dxx, such as Dxx_open. Therefore, on some systems, Dxx_init might simply be an empty function.

8.12 Opening Devices

Dxx_open opens a Dxx device and returns its status:

```
status = Dxx_open(device, name);
```

SIO_create calls Dxx_open to open a Dxx device. The following sequence of steps illustrates this process for an input-terminating device:



```
input = SIO_create("/adc16", SIO_INPUT, BUFSIZE, NULL)
```

1) Find string matching a prefix of **`/adc16`** in **`DEV_devtab[]`** device table. The associated **`DEV_Device`** structure contains driver functions, device ID, and device parameters.

2) Allocate **`DEV_Obj`** device object.

3) Assign **`bufsize`**, **`nbufs`**, **`segid`**, etc. fields in **`DEV_Obj`** from parameters and **`SIO_Attrs`** passed to **`SIO_create()`**.

4) Create **`todevice`** and **`fromdevice`** queues.

5) If opened for **`DEV_STANDARD`** streaming model, allocate **`attrs.nbufs`** buffers of size **`BUFSIZE`** and put them on **`todevice`** queue.

6) Call **`D`*`xx`*`_open()`** with pointer to new **`DEV_Obj`** and remaining name string.

```
status = Dxx_open(device, "16")
```

1) Validate fields in **`DEV_Obj`** pointed to by **`device`**.

2) Parse string for additional parameters (e.g., 16 kHz).

3) Allocate and initialize device-specific object.

4) Assign device-specific object to **`device->object`**.

The arguments to Dxx_open are:

```
DEV_Handle device;    /* driver handle */

String     name;      /* device name */
```

device points to an object of type DEV_Obj whose fields have been initialized by SIO_create. name is the string remaining after the device name has been matched by SIO_create using DEV_match.

Recall that SIO_create is called using the following syntax:

```
stream = SIO_create(name, mode, bufsize, attrs);
```

The name passed to SIO_create is typically a string indicating the device and an additional suffix, indicating some particular mode of operation of the device. For example, an analog-to-digital converter might have the base name /adc, while the sampling frequency might be indicated by a tag such as 16 for 16 kHz. The complete name passed to SIO_create would be /adc16.

SIO_create would identify the device by using DEV_match to match the string /adc against the list of configured devices. The string remainder 16 would be passed to Dxx_open to set the ADC to the correct sampling frequency.

Dxx_open usually allocates a device-specific object which is used to maintain the device state, as well as necessary semaphores. For a terminating device, this object typically has two SEM_Handle semaphore handles. One is used for synchronizing I/O operations (e.g., SIO_get, SIO_put, SIO_reclaim). The other handle is used with SIO_select to determine if a device is ready. A device object would typically be defined as:

```
typedef struct Dxx_Obj {
    SEM_Handle    sync;     /* synchronize I/O */
    SEM_Handle    ready;    /* used with SIO_select() */
    ‘other device-specific fields‘
} Dxx_obj, *Dxx_Handle;
```

The following template for Dxx_open shows the function’s typical features for a terminating device:

```
Int Dxx_open(DEV_Handle device, String name)
{
    Dxx_Handle objptr;

    /* check mode of device to be opened */
    if ( ‘device->mode is invalid‘ ) {
        return (SYS_EMODE);
    }
    /* check device id */
    if ( ‘device->devid is invalid‘ ) {
          return (SYS_ENODEV);
    }

    /* if device is already open, return error */
    if ( ‘device is in use‘ ) {
        return (SYS_EBUSY);
    }
    /* allocate device-specific object */
    objptr = MEM_alloc(0, sizeof (Dxx_Obj), 0);

    ‘fill in device-specific fields‘
    /*
     * create synchronization semaphore ...
     * select number of buffers based on device mode
     */
    objptr->sync = SEM_create( ‘number of buffers‘ , NULL);
    /* initialize ready semaphore for SIO_select()/Dxx_ready()
*/
    objptr->ready = NULL;

    ‘do any other device-specific initialization required‘

    /* assign initialized object */
    device->object = (Ptr)objptr;

    return (SYS_OK);
}
```

The first two steps take care of error checking. For example, a request to open an output-only device for input should generate an error message. A request to open channel ten on a five-channel system should also generate an error message.

The next step is to determine if the device is already opened. In many cases, an opened device cannot be re-opened, so a request to do so generates an error message.

If the device can be opened, the rest of Dxx_open consists of two major operations. First, the device-specific object is initialized, based in part on the device->params settings passed by SIO_create. Second, this object is attached to device->object. Dxx_open returns SYS_OK to SIO_create, which now has a properly initialized device object.

The configurable device parameters are used to set the operating parameters of the hardware. There are no DSP/BIOS constraints on which parameters should be set in Dxx_init rather than in Dxx_open.

objptr->sync is typically used to signal a task that is pending on the completion of an I/O operation. For example, a task may call SIO_put, which may block by pending on objptr->sync. When the required output is accomplished, SEM_post is called with objptr->sync. This makes a task blocked in Dxx_output ready to run.

DSP/BIOS does not impose any special constraints on the use of synchronization semaphores within a device driver. The appropriate use of such semaphores depends on the nature of the driver requirements and the underlying hardware.

The ready semaphore, objptr->ready, is used by Dxx_ready, which is called by SIO_select to determine if a device is available for I/O. This semaphore is explained in Section 4.6. Semaphores.

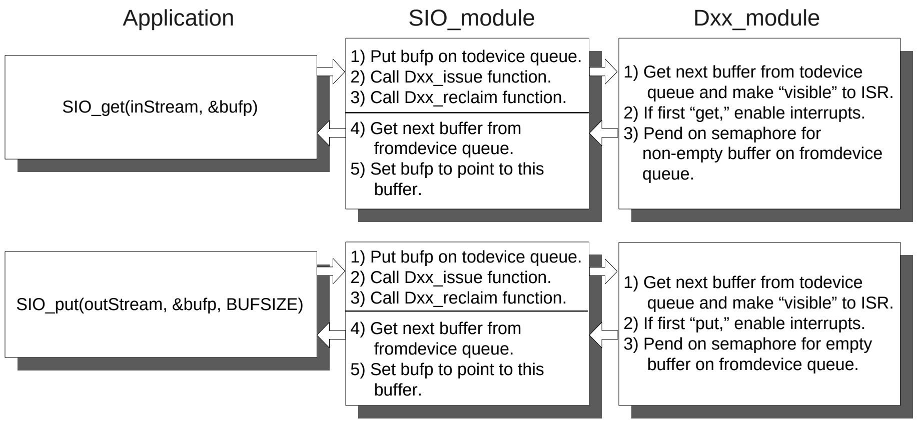
8.13 Real-time I/O

In DSP/BIOS there are two models that can be used for real-time I/O—the DEV_STANDARD streaming model and the DEV_ISSUERECLAIM streaming model. Each of these models is described in this section.

8.13.1 DEV_STANDARD Streaming Model

In the DEV_STANDARD streaming model, SIO_get is used to get a non-empty buffer from an input stream. To accomplish this, SIO_get first places an empty frame on the device->todevice queue. SIO_get then calls Dxx_issue, which starts the I/O and then calls Dxx_reclaim pending, until a full frame is available on the device->fromdevice queue. This blocking is accomplished by calling SEM_pend on the device semaphore objptr->sync that is posted whenever a buffer is filled.

Dxx_issue calls a low-level hardware function to initiate data input. When the required amount of data has been received, the frame is transferred to device->fromdevice. Typically, the hardware device will trigger an interrupt when a certain amount of data has been received. Dxx will handle this interrupt by means of an interrupt service routine (ISR), which will accumulate the data and determine if more data is needed for the waiting frame. If the ISR determines that the required amount of data has been received, the ISR will transfer the frame to device->fromdevice and then call SEM_post on the device semaphore. This will allow the task, blocked in Dxx_reclaim, to continue. Dxx_reclaim will then return to SIO_get, which will complete the input operation as illustrated.


Application
SIO_module
Dxx_module
SIO_get(inStream, &bufp)
1) Put bufp on todevice queue.
2) Call Dxx_issue function.
3) Call Dxx_reclaim function.
4) Get next buffer from fromdevice queue.
5) Set bufp to point to this buffer.
1) Get next buffer from todevice queue and make "visible" to ISR.
2) If first "get," enable interrupts.
3) Pend on semaphore for non-empty buffer on fromdevice queue.
SIO_put(outStream, &bufp, BUFSIZE)
1) Put bufp on todevice queue.
2) Call Dxx_issue function.
3) Call Dxx_reclaim function.
4) Get next buffer from fromdevice queue.
5) Set bufp to point to this buffer.
1) Get next buffer from todevice queue and make "visible" to ISR.
2) If first "put," enable interrupts.
3) Pend on semaphore for empty buffer on fromdevice queue.

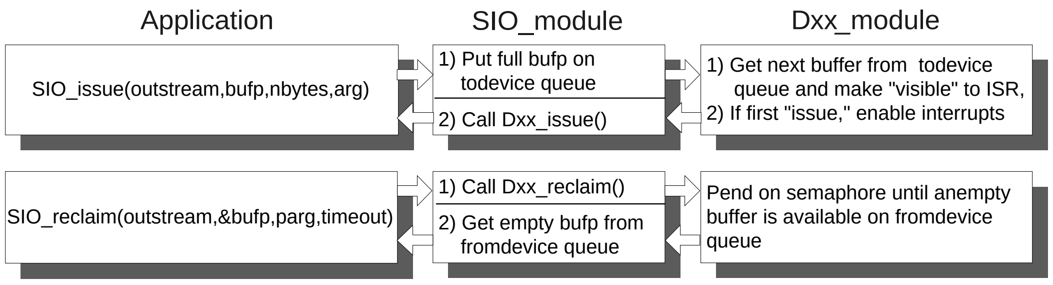
Note that objptr->sync is a counting semaphore and that tasks do not always block here. The value of objptr->sync represents the number of available frames on the fromdevice queue.

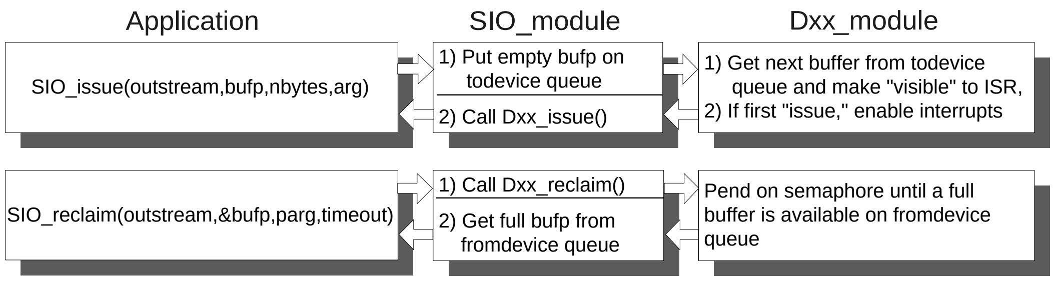
8.13.2 DEV_ISSUERECLAIM Streaming Model

In the DEV_ISSUERECLAIM streaming model SIO_issue is used to send buffers to a stream. To accomplish this, SIO_issue first places the frame on the device->todevice queue. It then calls Dxx_issue which starts the I/O and returns.

Dxx_issue calls a low-level hardware function to initialize I/O.

SIO_reclaim is used to retrieve buffers from the stream. This is done by calling Dxx_reclaim, which blocks until a frame is available on the device->fromdevice queue. This blocking is accomplished by calling SEM_pend on the device semaphore objptr->sync, just as for Dxx_issue. When the device ISR posts to objptr->sync, Dxx_reclaim is unblocked, and returns to SIO_reclaim. SIO_reclaim then gets the frame from the device->fromdevice queue and returns the buffer. This sequence is shown in the following two figures:


Application
SIO_module
Dxx_module
SIO_issue(outstream,bufp,nbytes,arg)
1) Put full bufp on todevice queue
2) Call Dxx_issue()
1) Get next buffer from todevice queue and make "visible" to ISR,
2) If first "issue," enable interrupts
SIO_reclaim(outstream,&bufp,parg,timeout)
1) Call Dxx_reclaim()
2) Get empty bufp from fromdevice queue
Pend on semaphore until anempty buffer is available on fromdevice queue


Application
SIO_module
Dxx_module
SIO_issue(outstream,bufp,nbytes,arg)
1) Put empty bufp on todevice queue
2) Call Dxx_issue()
1) Get next buffer from todevice queue and make "visible" to ISR,
2) If first "issue," enable interrupts
SIO_reclaim(outstream,&bufp,parg,timeout)
1) Call Dxx_reclaim()
2) Get full bufp from fromdevice queue
Pend on semaphore until a full buffer is available on fromdevice queue

Below is a template for Dxx_issue for a typical terminating device:

```
/*
 * ======== Dxx_issue ========
 */
Int Dxx_issue(DEV_Handle device)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;

    if ( ‘device is not operating in correct mode‘ ) {
        ‘start the device for correct mode‘
    }

    return (SYS_OK);
}
```

A call to Dxx_issue will start the device for the appropriate mode, either DEV_INPUT or DEV_OUTPUT. Once the device is known to be started, Dxx_issue simply returns. The actual data handling is performed by an ISR.

This is a template for Dxx_reclaim for a typical terminating device:

```
/*
 * ======== Dxx_reclaim ========
 */
Int Dxx_reclaim(DEV_Handle device)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;

    if (SEM_pend(objptr->sync, device->timeout)) {
       return (SYS_OK);
    }
    else {   /* SEM_pend() timed out */
       return (SYS_ETIMEOUT);
    }
}
```

A call to Dxx_reclaim will wait for the ISR to place a frame on the device->fromdevice queue, then return.

Dxx_reclaim calls SEM_pend with the timeout value specified at the time the stream is created (either by the Configuration Tool or with SIO_create) with this value. If the timeout expires before a buffer becomes available, Dxx_reclaim returns SYS_ETIMEOUT. In this situation, SIO_reclaim does not attempt to get anything from the device->fromdevice queue. SIO_reclaim returns SYS_ETIMEOUT, and does not return a buffer.

8.14 Closing Devices

A device is closed by calling SIO_delete, which in turn calls Dxx_idle and Dxx_close. Dxx_close closes the device after Dxx_idle returns the device to its initial state, which is the state of the device immediately after it was opened.

```
/*
 *  ======== Dxx_idle ========
 */
Int Dxx_idle(DEV_Handle device, Bool flush)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;
    Uns        post_count;

    /*
     * The only time we will wait for all pending data
     * is when the device is in output mode, and flush
     * was not requested.
     */
    if ((device->mode == DEV_OUTPUT) && !flush) {
        /* first, make sure device is started */
        if ( ‘device is not started‘ &&
             ‘device has received data‘ ) {
             ‘start the device‘
        }

        /*
         * wait for all output buffers to be consumed by the
         * output ISR. We need to maintain a count of how many
       * buffers are returned so we can set the semaphore later.
         */
        post_count = 0;
        while (!QUE_empty(device->todevice)) {
           SEM_pend(objptr->sync, SYS_FOREVER);
           post_count++;
        }

       if ( ‘there is a buffer currently in use by the ISR‘ ) {
            SEM_pend(objptr->sync, SYS_FOREVER);
            post_count++;
        }

        ‘stop the device‘

        /*
         * Don’t simply SEM_reset the count here. There is a
       * possibility that the ISR had just completed working on a
       * buffer just before we checked, and we don’t want to mess
         * up the semaphore count.
         */
        while (post_count > 0) {
            SEM_post(objptr->sync);
            post_count--;
        }
    }
else {  /* dev->mode = DEV_INPUT or flush was requested */
        ‘stop the device‘


        /*
        * do standard idling, place all frames in fromdevice
        * queue
        */
        while (!QUE_empty(device->todevice)) {
            QUE_put(device->fromdevice,
                 QUE_get(device->todevice));
            SEM_post(objptr->sync);
        }
    }

    return (SYS_OK);
}
```

The arguments to Dxx_idle are:

```
DEV_Handle  device;    /* driver handle */

Bool        flush;     /* flush indicator */
```

device is, as usual, a pointer to a DEV_Obj for this instance of the device. flush is a boolean parameter that indicates what to do with any pending data while returning the device to its initial state.

For a device in input mode, all pending data is always thrown away, since there is no way to force a task to retrieve data from a device. Therefore, the flush parameter has no effect on a device opened for input.

For a device opened for output, however, the flush parameter is significant. If flush is TRUE, any pending data is thrown away. If flush is FALSE, the Dxx_idle function will not return until all pending data has been rendered.

8.15 Device Control

Dxx_ctrl is called by SIO_ctrl to perform a control operation on a device. A typical use of Dxx_ctrl is to change the contents of a device control register or the sampling rate for an A/D or D/A device. Dxx_ctrl is called as follows:

```
status = Dxx_ctrl(DEV_Handle device, Uns cmd, Arg arg);
```

❑ cmd is a device-specific command.

❑ arg provides an optional command argument.

Dxx_ctrl returns SYS_OK if the control operation was successful; otherwise, Dxx_ctrl returns an error code.

8.16 Device Ready

Dxx_ready is called by SIO_select to determine if a device is ready for I/O. Dxx_ready returns TRUE if the device is ready and FALSE if the device is not. The device is ready if the next call to retrieve a buffer from the device will not block. This usually means that there is at least one available frame on the queue device->fromdevice when Dxx_ready returns. Refer to Section 8.6, page 8-25, Selecting Among Multiple Streams, for more information on SIO_select.

```
Bool Dxx_ready(DEV_Handle dev, SEM_Handle sem)
{
    Dxx_Handle objptr = (Dxx_Handle)device->object;

    /* register the ready semaphore */
    objptr->ready = sem;

    if ((device->mode == DEV_INPUT) &&
       ((device->model == DEV_STANDARD) &&
        ‘device is not started‘ )) {
        ‘start the device‘
    }

    /* return TRUE if device is ready */
    return ( ‘TRUE if device->fromdevice has a frame or
           device won’t block‘ );
}
```

If the mode is DEV_INPUT, the streaming model is DEV_STANDARD, and the device has not been started, the device is started. This is necessary, since in the DEV_STANDARD streaming model, SIO_select may be called by the application before the first call to SIO_get.

The device’s ready semaphore handle is set to the semaphore handle passed in by SIO_select. To better understand Dxx_ready, consider the following details of SIO_select.

SIO_select can be summarized in pseudocode as follows:

```
/*
 *  ======== SIO_select ========
 */
Uns SIO_select(streamtab, n, timeout)
    SIO_Handle streamtab[];    /* array of streams */
    Int        n;              /* number of streams */
    Uns        timeout;        /*   passed to SEM_pend() */
{
    Int        i;
    Uns        mask = 1;       /* used to build ready mask */
    Uns        ready = 0;      /* bit mask of ready streams */
    SEM_Handle sem;            /* local semaphore */
    SIO_Handle *stream;        /* pointer into streamtab[] */
    /*
     * For efficiency, the "real" SIO_select() doesn’t call
     * SEM_create() but instead initializes a SEM_Obj on the
     * current stack.
     */
    sem = SEM_create(0, NULL);

    stream = streamtab;

    for (i = n; i > 0; i--, stream++) {
        /*
         * call each device ready function with ’sem’
         */
        if ( ‘Dxx_ready(device, sem)‘ )
            ready = 1;
        }
    }
    if (!ready) {
        /* wait until at least one device is ready */
        SEM_pend(sem, timeout);
    }
    ready = 0;

    stream = streamtab;

    for (i = n; i > 0; i--, stream++) {
        /*
         * Call each device ready function with NULL.
         * When this loop is done, ready will have a bit set
         * for each ready device.
         */
        if ( ‘Dxx_ready(device, NULL)‘ )
            ready |= mask;
        }
        mask = mask << 1;
    }

    return (ready);
}
```

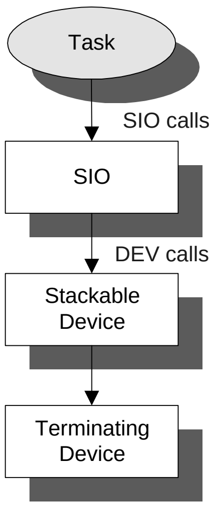
SIO_select makes two calls to Dxx_ready for each Dxx device. The first call is used to “register” sem with the device, and the second call (with sem = NULL) is used to “un-register” sem.

Each Dxx_ready function holds on to sem in its device-specific object (e.g., objptr->ready = sem). When an I/O operation completes (i.e., a buffer has been filled or emptied), and objptr->ready is not NULL, SEM_post is called to post objptr->ready.

If at least one device is ready, or if SIO_select was called with timeout equal to 0, SIO_select will not block; otherwise, SIO_select will pend on the ready semaphore until at least one device is ready, or until the time-out has expired.

Consider the case where a device becomes ready before a time-out occurs. The ready semaphore is posted by whichever device becomes ready first. SIO_select then calls Dxx_ready again for each device, this time with sem = NULL. This has two effects. First, any additional Dxx device that becomes ready will not post the ready semaphore. This prevents devices from posting to a semaphore that no longer exists, since the ready semaphore is maintained in the local memory of SIO_select. Second, by polling each device a second time, SIO_select can determine which devices have become ready since the first call to Dxx_ready, and set the corresponding bits for those devices in the ready mask.

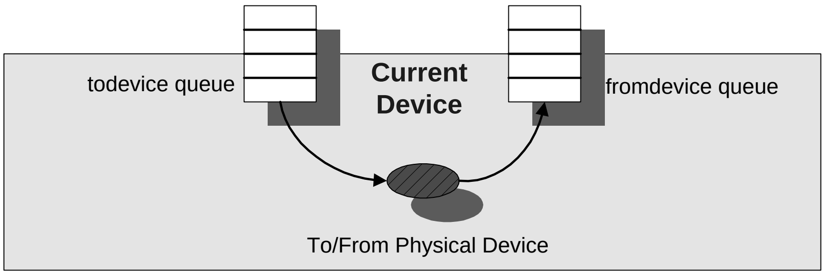
8.17 Types of Devices



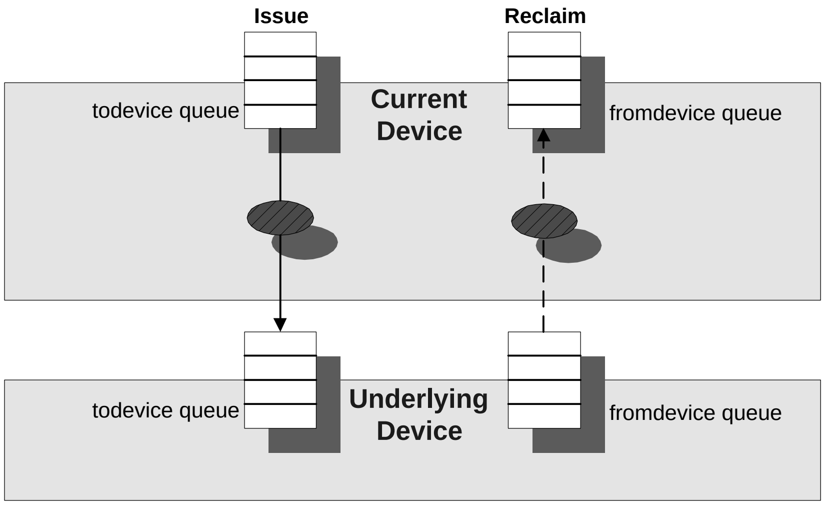
There are two main types of devices: terminating devices and stackable devices. Each exports the same device functions, but they implement them slightly differently. A terminating device is any device that is a data source or sink. A stackable device is any device that does not source or sink data, but uses the DEV functions to send (or receive) data to or from another device. Refer to the figure at right to see how the stacking and terminating devices fit into a stream.

Within the broad category of stackable devices, there are two distinct types. These are referred to as in-place stacking devices and copying stacking devices. The in-place stacking device performs in-place manipulations on data in buffers. The copying stacking device moves the data to another buffer while processing the data. Copying is necessary for devices that produce more data than they receive (for example, an unpacking device or an audio decompression driver), or because they require access to the whole buffer to generate output samples and cannot overwrite their input data (for example, an FFT driver). These types of stacking devices require different implementation, since the copying device may have to supply its own buffers.

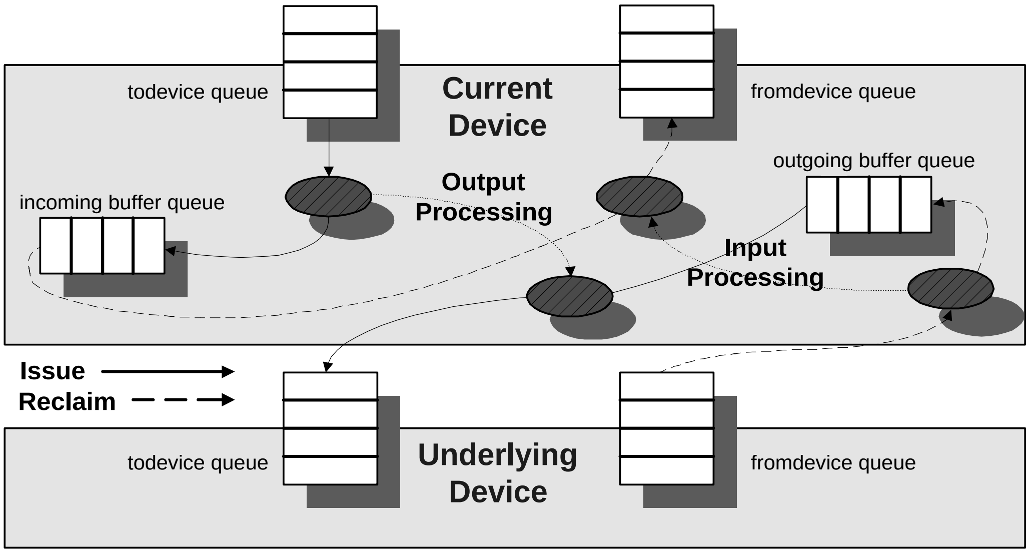
The figure below shows the buffer flow of a typical terminating device. The interaction with DSP/BIOS is relatively simple. Its main complexities exist in the code to control and stream data to and from the physical device.



The diagram below shows the buffer flow of an “in-place” stacking driver. Notice that all data processing is done in a single buffer. This is a relatively simple device, but it is not as general-purpose as the copying stacking driver.



The figure below shows the buffer flow of a copying stacking driver. Notice that the buffers that come down from the task side of the stream never actually move to the device side of the stream. The two buffer pools remain independent. This is important, since in a copying stacking device, the task-side buffers may be a different size than the device-side buffers. Also, care is taken to preserve the order of the buffers coming into the device, so the SIO_ISSUERECLAIM streaming model can be supported.



Index