

# ***TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide***

Literature Number: SPRU404C  
April 2001



Printed on Recycled Paper

## Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

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

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

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

Mailing Address:

Texas Instruments  
Post Office Box 655303  
Dallas, Texas 75265

# Read This First

---

---

---

### **About This Manual**

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320C5000™ DSP devices 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 *TMS320 DSP/BIOS User's Guide*, a companion volume to this API reference guide.

Before you read this manual, you may use the "Using DSP/BIOS" lessons in the online *Code Composer Studio Tutorial* and the DSP/BIOS section of the online help 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 DSP/BIOS.

### **Notational Conventions**

This document uses the following conventions:

- ❑ 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.
- ❑ Throughout this manual, 54 represents the two-digit numeric appropriate to your specific DSP platform. For example, DSP/BIOS assembly language API header files for the C6000 platform are described as having a suffix of .h54. For the C55 DSP platform, substitute 55 for each occurrence of 54.
- ❑ Information specific to a particular device is designated with one of the following icons:



## **Related Documentation From Texas Instruments**

The following books describe TMS320 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.

**TMS320C54x Assembly Language Tools User's Guide** (literature number SPRU102) 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 C5000 generation of devices.

**TMS320C55x Assembly Language Tools User's Guide** (literature number SPRU280) 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 C5000 generation of devices.

**TMS320C54x Optimizing C Compiler User's Guide** (literature number SPRU103) describes the C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the C54x generation of devices.

**TMS320C55x Optimizing C Compiler User's Guide** (literature number SPRU281) describes the C55x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the C55x generation of devices.

**TMS320C55x Programmer's Guide** (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x DSPs and includes application program examples.

**TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals** (literature number SPRU131) describes the TMS320C54x 16-bit fixed-point general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

**TMS320C54x DSP Enhanced Peripherals Ref Set, Vol 5** (literature number SPRU302) describes the enhanced peripherals available on the TMS320C54x digital signal processors. Includes the multi channel buffered serial ports (McBSPs), direct memory access (DMA) controller, interprocessor communications, and the HPI-8 and HPI-16 host port interfaces.

**TMS320C54x DSP Mnemonic Instruction Set Reference Set Volume 2** (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

**TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set** (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

**TMS320C54x Code Composer Studio Tutorial Online Help** (literature number SPRH134) introduces the Code Composer Studio integrated development environment and software tools. Of special interest to DSP/BIOS users are the *Using DSP/BIOS* lessons.

**TMS320C55x Code Composer Studio Tutorial Online Help** (literature number SPRH097) introduces the Code Composer Studio integrated development environment and software tools. Of special interest to DSP/BIOS users are the *Using DSP/BIOS* lessons.

**Code Composer Studio Application Program Interface (API) Reference Guide** (literature number SPRU321) describes the Code Composer Studio application programming interface, which allows you to program custom analysis tools for Code Composer Studio.

**DSP/BIOS and TMS320C54x Extended Addressing** (literature number SPRA599) provides basic run-time services including real-time analysis functions for instrumenting an application, clock and periodic functions, I/O modules, and a preemptive scheduler. It also describes the far model for extended addressing, which is available on the TMS320C54x platform.

## **Related Documentation**

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

***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

***Programming Embedded Systems in C and C++***, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

***Real-Time Systems***, by Jane W. S. Liu, published by Prentice Hall; ISBN: 013099651, June 2000

***Principles of Concurrent and Distributed Programming*** (Prentice Hall International Series in Computer Science), by M. Ben-Ari, published by Prentice Hall; ISBN: 013711821X, May 1990

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

## **Trademarks**

MS-DOS, Windows, and Windows NT are trademarks of Microsoft Corporation.

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, XDS, Code Composer, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, BIOSuite, SPOX, TMS320, TMS320C54x, TMS320C55x, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C5000, and TMS320C6000.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

# Contents

---

---

---

<b>1</b>	<b>API Functional Overview</b>	<b>1-1</b>
	<i>This chapter provides an overview to the TMS320C5000™ DSP/BIOS API functions.</i>	
1.1	DSP/BIOS Modules	1-2
1.2	Naming Conventions	1-3
1.3	List of Operations	1-4
<b>2</b>	<b>Application Program Interface</b>	<b>2-1</b>
	<i>This chapter describes the TMS320C5000™ DSP/BIOS API functions, which are alphabetized by name. In addition, there are reference sections that describe the overall capabilities of each module.</i>	
2.1	Assembly Language Interface Overview	2-2
2.2	ATM Module	2-4
2.3	C54 and C55 Modules	2-17
2.4	CLK Module	2-26
2.5	DEV Module	2-36
2.6	Global Settings	2-74
2.7	HST Module	2-76
2.8	HWI Module	2-81
2.9	IDL Module	2-103
2.10	LCK Module	2-106
2.11	LOG Module	2-111
2.12	MBX Module	2-127
2.13	MEM Module	2-133
2.14	PIP Module	2-148
2.15	PRD Module	2-169
2.16	QUE Module	2-180
2.17	RTDX Module	2-197
2.18	SEM Module	2-214
2.19	SIO Module	2-224
2.20	STS Module	2-244
2.21	SWI Module	2-258
2.22	SYS Module	2-297
2.23	TRC Module	2-312
2.24	TSK Module	2-321
2.25	std.h and stdlib.h functions	2-356

<b>3</b>	<b>Utility Programs</b>	<b>3-1</b>
	<i>This chapter provides documentation for TMS320C5000 utilities that can be used to examine various files from the MS-DOS command line. These programs are provided with DSP/BIOS in the bin subdirectory. Any other utilities that may occasionally reside in the bin subdirectory and not documented here are for internal Texas Instruments' use only.</i>	
3.1	cdbprint	3-2
3.2	gconfgen	3-3
3.3	nmti	3-7
3.4	sectti	3-8
3.5	sizeti	3-9
3.6	vers	3-10
<b>A</b>	<b>Function Callability and Error Tables</b>	<b>A-1</b>
	<i>This appendix provides tables describing TMS320C5000™ errors and function callability.</i>	
A.1	Functions Callable by Tasks, SWI Handlers, or Hardware ISRs	A-2
A.2	DSP/BIOS Error Codes	A-8
<b>B</b>	<b>C55x DSP/BIOS Register Use and Preservation Conventions</b>	<b>B-1</b>
	<i>This appendix provides tables describing the TMS320C55x™ register conventions in terms of usage and preservation across multi-threaded context switching.</i>	
B.1	Preservation Model for Non-Status CPU Registers	B-2
B.2	Preservation Model for Processor Status Registers	B-5



# Figures

---

---

---

---

2-1	RTA Control Panel Properties Page .....	2-113
2-2	Pipe Schematic .....	2-149
2-3	PRD Tick Cycles .....	2-175
2-4	Statistics Accumulation on the Host.....	2-247
2-5	RTA Control Panel Properties Page.....	2-249
2-6	RTA Control Panel Properties Page .....	2-314

---

# Tables

---

---

---

---

1-1	DSP/BIOS Modules .....	1-2
1-2	The DSP/BIOS Operations .....	1-4
2-1	HWI interrupts for the C54x .....	2-86
2-2	HWI interrupts for the 'C55x .....	2-87
2-3	Conversion Characters for LOG_printf .....	2-123
2-4	Typical Memory Segments .....	2-140
2-5	Statistics Units for HWI, PIP, PRD, and SWI Modules .....	2-245
2-6	Conversion Characters Recognized by SYS_printf .....	2-303
2-7	Conversion Characters Recognized by SYS_sprintf .....	2-305
2-8	Conversion Characters Recognized by SYS_vprintf .....	2-307
2-9	Conversion Characters Recognized by SYS_vsprintf.....	2-309
2-10	Events and Statistics Traced by TRC .....	2-312
A.1	Functions Callable by Tasks, SWI Handlers, or Hardware ISRs .....	A-2
A.2	DSP/BIOS Error Codes .....	A-8
B.1	Preservation Model for Non-Status CPU Registers.....	B-2
B.2	Preservation Model for Procesor Status Registers.....	B-5

# API Functional Overview

---

---

---

This chapter provides an overview to the TMS320C5000™ DSP/BIOS API functions.

Topic	Page
1.1 DSP/BIOS Modules .....	1-2
1.2 Naming Conventions.....	1-3
1.3 List of Operations .....	1-4

## 1.1 DSP/BIOS Modules

Table 1-1. DSP/BIOS Modules

Module	Description
ATM	Atomic functions written in assembly language
C54	Target-specific functions
C55	Target-specific functions
CLK	System clock manager
DEV	Device driver interface
GBL	Global setting manager
HST	Host channel manager
HWI	Hardware interrupt manager
IDL	Idle function and processing loop manager
LCK	Resource lock manager
LOG	Event Log manager
MBX	Mailboxes manager
MEM	Memory manager
PIP	Buffered pipe manager
PRD	Periodic function manager
QUE	Queue manager
RTDX	Real-time data exchange manager
SEM	Semaphores manager
SIO	Stream I/O manager
STS	Statistics object manager
SWI	Software interrupt manager
SYS	System services manager
TRC	Trace manager
TSK	Multitasking manager
C library stdlib.h	Standard C library I/O functions

## 1.2 Naming Conventions

The format for a DSP/BIOS operation name is a 3- or 4-letter prefix for the module that contains the operation, an underscore, and the action.

In the Assembly Interface section for each macro, Preconditions lists registers that must be set before using the macro. Postconditions lists the registers set by the macro that you may want to use. Modifies lists all individual registers modified by the macro, including registers in the Postconditions list. Several macros modify a 32-bit register. In these cases, the Modifies list includes both the high and low registers that make up the 32-bit register.

### 1.3 List of Operations

Table 1-2. The DSP/BIOS Operations

a. ATM module operations

Function	Operation
ATM_andi, ATM_andu	Atomically AND two memory locations and return previous value of the second
ATM_cleari, ATM_clearu	Atomically clear memory location and return previous value
ATM_deci, ATM_decu	Atomically decrement memory and return new value
ATM_inci, ATM_incu	Atomically increment memory and return new value
ATM_ori, ATM_oru	Atomically OR memory location and return previous value
ATM_seti, ATM_setu	Atomically set memory and return previous value

b. C54/55 operations

Function	Operation
C54_disableIMR	Disable certain maskable interrupts
C54_enableIMR	Enable certain maskable interrupts
C55_disableIEMR0 C55_disableIER1	Disable certain maskable interrupts
C55_enableIEMR0 C55_enableIER1	Enable certain maskable interrupts
C55_plug	C function to plug an interrupt vector
C54_plug	C function to plug an interrupt vector

c. CLK module operations

Function	Operation
CLK_countspms	Number of hardware timer counts per millisecond
CLK_gettime	Get high-resolution time
CLK_gettime	Get low-resolution time
CLK_getprd	Get period register value

## d. DEV module operations

Function	Operation
DEV_match	Match a device name with a driver
Dxx_close	Close device
Dxx_ctrl	Device control operation
Dxx_idle	Idle device
Dxx_init	Initialize device
Dxx_issue	Send a buffer to the device
Dxx_open	Open device
Dxx_ready	Check if device is ready for I/O
Dxx_reclaim	Retrieve a buffer from a device

## e. HST module operations

Function	Operation
HST_getpipe	Get corresponding pipe object

## f. HWI module operations

Function	Operation
HWI_disable	Globally disable hardware interrupts
HWI_enable	Globally enable hardware interrupts
HWI_enter	Hardware interrupt service routine prolog
HWI_exit	Hardware interrupt service routine epilog
HWI_restore	Restore global interrupt enable state

## g. IDL module operations

Function	Operation
IDL_run	Make one pass through idle functions

h. LCK module operations

Function	Operation
LCK_create	Create a resource lock
LCK_delete	Delete a resource lock
LCK_pend	Acquire ownership of a resource lock
LCK_post	Relinquish ownership of a resource lock

i. LOG module operations

Function	Operation
LOG_disable	Disable a log
LOG_enable	Enable a log
LOG_error/LOG_message	Write a message to the system log
LOG_event	Append an unformatted message to a log
LOG_printf	Append a formatted message to a message log
LOG_reset	Reset a log

j. MBX module operations

Function	Operation
MBX_create	Create a mailbox
MBX_delete	Delete a mailbox
MBX_pend	Wait for a message from mailbox
MBX_post	Post a message to mailbox



## k. The MEM module operations:

Function	Operation
MEM_alloc, MEM_valloc, MEM_calloc	Allocate from a memory heap
MEM_define	Define a new memory heap
MEM_free	Free a block of memory
MEM_redefine	Redefine an existing memory heap
MEM_stat	Return the status of a memory heap

## l. PIP module operations

Function	Operation
PIP_alloc	Get an empty frame from a pipe
PIP_free	Recycle a frame that has been read back into a pipe
PIP_get	Get a full frame from a pipe
PIP_getReaderAddr	Get the value of the readerAddr pointer of the pipe
PIP_getReaderNumFrames	Get the number of pipe frames available for reading
PIP_getReaderSize	Get the number of words of data in a pipe frame
PIP_getWriterAddr	Get the value of the writerAddr pointer of the pipe
PIP_getWriterNumFrames	Get the number of pipe frames available to be written to
PIP_getWriterSize	Get the number of words that can be written to a pipe frame
PIP_peek	Get the pipe frame size and address without actually claiming the pipe frame
PIP_put	Put a full frame into a pipe
PIP_reset	Reset all fields of a pipe object to their original values
PIP_setWriterSize	Set the number of valid words written to a pipe frame

## m. PRD module operations

Function	Operation
PRD_getticks	Get the current tick counter
PRD_start	Arm a periodic function for one-time execution
PRD_stop	Stop a periodic function from execution
PRD_tick	Advance tick counter, dispatch periodic functions

n. QUE module operations

Function	Operation
QUE_create	Create an empty queue
QUE_delete	Delete an empty queue
QUE_dequeue	Remove from front of queue (non-atomically)
QUE_empty	Test for an empty queue
QUE_enqueue	Insert at end of queue (non-atomically)
QUE_get	Get element from front of queue (atomically)
QUE_head	Return element at front of queue
QUE_insert	Insert in middle of queue (non-atomically)
QUE_new	Set a queue to be empty
QUE_next	Return next element in queue (non-atomically)
QUE_prev	Return previous element in queue (non-atomically)
QUE_put	Put element at end of queue (atomically)
QUE_remove	Remove from middle of queue (non-atomically)

## o. RTDX module operations

Function	Operation
RTDX_channelBusy	Return status indicating whether a channel is busy
RTDX_CreateInputChannel	Declare input channel structure
RTDX_CreateOutputChannel	Declare output channel structure
RTDX_disableInput	Disable an input channel
RTDX_disableOutput	Disable an output channel
RTDX_enableInput	Enable an input channel
RTDX_enableOutput	Enable an output channel
RTDX_isInputEnabled	Return status of the input data channel
RTDX_isOutputEnabled	Return status of the output data channel
RTDX_read	Read from an input channel
RTDX_readNB	Read from an input channel without blocking
RTDX_sizeofInput	Return the number of bytes read from an input channel
RTDX_write	Write to an output channel

## p. SEM module operations

Function	Operation
SEM_count	Get current semaphore count
SEM_create	Create a semaphore
SEM_delete	Delete a semaphore
SEM_ipost	Signal a semaphore (interrupt only)
SEM_new	Initialize a semaphore
SEM_pend	Wait for a semaphore
SEM_post	Signal a semaphore
SEM_reset	Reset semaphore

q. SIO module operations

Function	Operation
SIO_bufsize	Size of the buffers used by a stream
SIO_create	Create stream
SIO_ctrl	Perform a device-dependent control operation
SIO_delete	Delete stream
SIO_flush	Idle a stream by flushing buffers
SIO_get	Get buffer from stream
SIO_idle	Idle a stream
SIO_issue	Send a buffer to a stream
SIO_put	Put buffer to a stream
SIO_reclaim	Request a buffer back from a stream
SIO_segid	Memory section used by a stream
SIO_select	Select a ready device
SIO_staticbuf	Acquire static buffer from stream

r. STS module operations

Function	Operation
STS_add	Add a value to a statistics object
STS_delta	Add computed value of an interval to object
STS_reset	Reset the values stored in an STS object
STS_set	Store initial value of an interval to object

## s. SWI module operations

Function	Operation
SWI_andn	Clear bits from SWI's mailbox and post if becomes 0
SWI_andnHook	Specialized version of SWI_andn
SWI_create	Create a software interrupt
SWI_dec	Decrement SWI's mailbox and post if becomes 0
SWI_delete	Delete a software interrupt
SWI_disable	Disable software interrupts
SWI_enable	Enable software interrupts
SWI_getattr	Get attributes of a software interrupt
SWI_getmbx	Return SWI's mailbox value
SWI_getpri	Return an SWI's priority mask
SWI_inc	Increment SWI's mailbox and post
SWI_or	Set or mask in an SWI's mailbox and post
SWI_oHook	Specialized version of SWI_or
SWI_post	Post a software interrupt
SWI_raisepri	Raise an SWI's priority
SWI_restorepri	Restore an SWI's priority
SWI_self	Return address of currently executing SWI object
SWI_setattr	Set attributes of a software interrupt

## t. SYS module operations

Function	Operation
SYS_abort	Abort program execution
SYS_atexit	Stack an exit handler
SYS_error	Flag error condition
SYS_exit	Terminate program execution
SYS_printf, SYS_sprintf, SYS_vprintf, SYS_vsprintf	Formatted output
SYS_putchar	Output a single character

u. TRC module operations

Function	Operation
TRC_disable	Disable a set of trace controls
TRC_enable	Enable a set of trace controls
TRC_query	Test whether a set of trace controls is enabled

v. TSK module operations

Function	Operation
TSK_checkstacks	Check for stack overflow
TSK_create	Create a task ready for execution
TSK_delete	Delete a task
TSK_deltatime	Update task STS with time difference
TSK_disable	Disable DSP/BIOS task scheduler
TSK_enable	Enable DSP/BIOS task scheduler
TSK_exit	Terminate execution of the current task
TSK_getenv	Get task environment
TSK_geterr	Get task error number
TSK_getname	Get task name
TSK_getpri	Get task priority
TSK_getsts	Get task STS object
TSK_itick	Advance system alarm clock (interrupt only)
TSK_self	Returns a handle to the current task
TSK_setenv	Set task environment
TSK_seterr	Set task error number
TSK_setpri	Set a task execution priority
TSK_settime	Set task STS previous time
TSK_sleep	Delay execution of the current task
TSK_stat	Retrieve the status of a task
TSK_tick	Advance system alarm clock
TSK_tick	Return current value of system clock
TSK_tick	Yield processor to equal priority task

## w. C library stdlib.h

Function	Operation
atexit	Registers one or more exit functions used by exit
calloc	Allocates memory block initialized with zeros
exit	Calls the exit functions registered in atexit
free	Frees memory block
getenv	Searches for a matching environment string
malloc	Allocates memory block
realloc	Resizes previously allocated memory block

## x.) DSP/BIOS std.h special utility C macros

Function	Operation
ArgToInt(arg)	Casting to treat Arg type parameter as integer (Int) type on the given target
ArgToPtr(arg)	Casting to treat Arg type parameter as pointer (Ptr) type on the given target

# Application Program Interface

This chapter describes the TMS320C5000™ DSP/BIOS API functions, which are alphabetized by name. In addition, there are reference sections that describe the overall capabilities of each module.

Topic	Page
2.1 Assembly Language Interface Overview . . . . .	2-2
2.2 ATM Module . . . . .	2-4
2.3 C54 and C55 Modules . . . . .	2-17
2.4 CLK Module . . . . .	2-26
2.5 DEV Module . . . . .	2-36
2.6 Global Settings . . . . .	2-74
2.7 HST Module . . . . .	2-76
2.8 HWI Module . . . . .	2-81
2.9 IDL Module . . . . .	2-103
2.10 LCK Module . . . . .	2-106
2.11 LOG Module . . . . .	2-111
2.12 MBX Module . . . . .	2-127
2.13 MEM Module . . . . .	2-133
2.14 PIP Module . . . . .	2-148
2.15 PRD Module . . . . .	2-169
2.16 QUE Module . . . . .	2-180
2.17 RTDX Module . . . . .	2-197
2.18 SEM Module . . . . .	2-214
2.19 SIO Module . . . . .	2-224
2.20 STS Module . . . . .	2-244
2.21 SWI Module . . . . .	2-258
2.22 SYS Module . . . . .	2-297
2.23 TRC Module . . . . .	2-312
2.24 TSK Module . . . . .	2-321
2.25 std.h and stdlib.h functions . . . . .	2-356



## 2.1 Assembly Language Interface Overview

When calling DSP/BIOS APIs from assembly source code, you should include the module.h54 or module.h6455 header file for any API modules used. This modular approach reduces the assembly time of programs that do not use all the modules.

Where possible, you should use the DSP/BIOS API macros instead of using assembly instructions directly. The DSP/BIOS API macros provide a portable, optimized way to accomplish the same task. For example, use HWI\_disable instead of the equivalent instruction to temporarily disable interrupts. On some devices, disabling interrupts in a threaded interface is more complex than it appears. Some of the DSP/BIOS API functions have assembly macros and some do not.

Most of the DSP/BIOS API macros do not have parameters. Instead they expect parameter values to be stored in specific registers when the API macro is called. This makes your program more efficient. A few API macros accept constant values as parameters. For example, HWI\_enter and HWI\_exit accept constants defined as bitmasks identifying the registers to save or restore.

The **Preconditions** section for each DSP/BIOS API macro in this chapter lists registers that must be set before using the macro.

The **Postconditions** section lists registers set by the macro.

**Modifies** lists all individual registers modified by the macro, including registers in the Postconditions list.

### Example

#### Assembly Interface



Syntax	SWI_getpri
Preconditions	ar2 = address of the SWI object
Postconditions	a = SWI object's priority mask
Modifies	ag, ah, al, c

## Assembly Interface



### Syntax

SWI\_getpri

### Preconditions

xar0 = address of the SWI object

### Postconditions

t0 = SWI object's priority mask

### Modifies

t0

Assembly functions can call C functions. Remember that the C compiler adds an underscore prefix to function names, so when calling a C function from assembly, add an underscore to the beginning of the C function name. For example, call `_myfunction` instead of `myfunction`. See the *TMS320C54x Optimizing Compiler User's Guide* or *TMS320C55x Optimizing Compiler User's Guide* for more details.

The Configuration Tool creates two names for each object: one beginning with an underscore, and one without. This allows you to use the name without the underscore in both C and assembly language functions.

All BIOS APIs are preconditioned per standard C conventions. Individual APIs in this document only indicate additional conditions, if any.

BIOS APIs save/restore context for each task during the context switch that comprises all the registers listed as *Save by Child* in the C compiler manual appropriate for your platform. You must save/restore all additional register context you chose to manipulate directly in assembly or otherwise.



The large memory model (see GBL module, Page 2–75, for more detail) is the default while listing registers in the **Precondition**, **Postcondition** and **Modifies** sections of API descriptions. This means references to `xarx`, `xssp`, `xssp`, etc., are extended registers rather than their 16-bit variants such as `arx` and `sp`. In the small model, you can assume the upper bits of the extended register are designated as *don't care* unless explicitly indicated as an *exception*.

## 2.2 ATM Module

The ATM module includes assembly language functions.

### Functions

- ❑ ATM\_andi, ATM\_andu. AND memory and return previous value
- ❑ ATM\_cleari, ATM\_clearu. Clear memory and return previous value
- ❑ ATM\_deci, ATM\_decu. Decrement memory and return new value
- ❑ ATM\_inci, ATM\_incu. Increment memory and return new value
- ❑ ATM\_ori, ATM\_oru. OR memory and return previous value
- ❑ ATM\_seti, ATM\_setu. Set memory and return previous value

### Description

ATM provides a set of assembly language functions that are used to manipulate variables with interrupts disabled. These functions can therefore be used on data shared between tasks, and on data shared between tasks and interrupt routines.

**ATM\_andi***Atomically AND Int memory location and return previous value***C Interface****Syntax**

ival = ATM\_andi(idst, isrc);

**Parameters**

volatile Int	*idst;	/* pointer to integer */
Int	isrc;	/* integer mask */

**Return Value**

Int	ival;	/* previous value of *idst */
-----	-------	-------------------------------

**Assembly Interface**

none

**Description**

ATM\_andi atomically ANDs the mask contained in isrc with a destination memory location and overwrites the destination value \*idst with the result as follows:

```
`interrupt disable`
ival = *idst;
*idst = ival & isrc;
`interrupt enable`
return(ival);
```

ATM\_andi is written in assembly language, efficiently disabling interrupts on the target processor during the call.

**See Also**

ATM\_andu  
ATM\_ori

ATM\_andu

Atomically AND Uns memory location and return previous value

C Interface

Syntax	uval = ATM_andu(udst, usrc);
Parameters	volatile Uns *udst; /* pointer to unsigned */ Uns usrc; /* unsigned mask */
Return Value	Uns uval; /* previous value of *udst */

Assembly Interface

none

**Description** ATM\_andu atomically ANDs the mask contained in usrc with a destination memory location and overwrites the destination value \*udst with the result as follows:

```
`interrupt disable`  
uval = *udst;  
*udst = uval & usrc;  
`interrupt enable`  
return(uval);
```

ATM\_andu is written in assembly language, efficiently disabling interrupts on the target processor during the call.

**See Also** ATM\_andi  
ATM\_oru

**ATM\_cleari***Atomically clear Int memory location and return previous value***C Interface**

<b>Syntax</b>	<code>ival = ATM_cleari(idst);</code>
<b>Parameters</b>	<code>volatile Int   *idst;    /* pointer to integer */</code>
<b>Return Value</b>	<code>Int            ival;    /* previous value of *idst */</code>

**Assembly Interface**       none

**Description**           ATM\_cleari atomically clears an Int memory location and returns its previous value as follows:

```
`interrupt disable`  
ival = *idst;  
*dst = 0;  
`interrupt enable`  
return (ival);
```

ATM\_cleari is written in assembly language, efficiently disabling interrupts on the target processor during the call.

**See Also**               ATM\_clearu  
                          ATM\_seti

ATM\_clearu

Atomically clear Uns memory location and return previous value

C Interface

Syntax	uval = ATM_clearu(udst);
Parameters	volatile Uns *udst; /* pointer to unsigned */
Return Value	Uns uval; /* previous value of *udst */
Assembly Interface	none
Description	<p>ATM_clearu atomically clears an Uns memory location and returns its previous value as follows:</p> <pre>`interrupt disable` uval = *udst; *udst = 0; `interrupt enable` return (uval);</pre> <p>ATM_clearu is written in assembly language, efficiently disabling interrupts on the target processor during the call.</p>
See Also	ATM_clearu ATM_setu

**ATM\_deci***Atomically decrement Int memory and return new value***C Interface**

<b>Syntax</b>	<code>ival = ATM_deci(idst);</code>
<b>Parameters</b>	<code>volatile Int   *idst;    /* pointer to integer */</code>
<b>Return Value</b>	<code>Int            ival;       /* new value after decrement */</code>

**Assembly Interface**

none

**Description**

ATM\_deci atomically decrements an Int memory location and returns its new value as follows:

```
`interrupt disable`
ival = *idst - 1;
*idst = ival;
`interrupt enable`
return (ival);
```

ATM\_deci is written in assembly language, efficiently disabling interrupts on the target processor during the call.

Decrementing a value equal to the minimum signed integer results in a value equal to the maximum signed integer.

**See Also**

ATM\_decu  
ATM\_inci



<b>ATM_decu</b>	<i>Atomically decrement Uns memory and return new value</i>
<b>C Interface</b>	
<b>Syntax</b>	uval = ATM_decu(udst);
<b>Parameters</b>	volatile Uns *udst; /* pointer to unsigned */
<b>Return Value</b>	Uns uval; /* new value after decrement */
<b>Assembly Interface</b>	none
<b>Description</b>	<p>ATM_decu atomically decrements a Uns memory location and returns its new value as follows:</p> <pre>`interrupt disable` uval = *udst - 1; *udst = uval; `interrupt enable` return (uval);</pre> <p>ATM_decu is written in assembly language, efficiently disabling interrupts on the target processor during the call.</p> <p>Decrementing a value equal to the minimum unsigned integer results in a value equal to the maximum unsigned integer.</p>
<b>See Also</b>	ATM_deci ATM_incu

**ATM\_inci***Atomically increment Int memory and return new value***C Interface****Syntax**`ival = ATM_inci(idst);`**Parameters**`volatile Int    *idst;    /* pointer to integer */`**Return Value**`Int            ival;       /* new value after increment */`**Assembly Interface**

none

**Description**

ATM\_inci atomically increments an Int memory location and returns its new value as follows:

```
`interrupt disable`  
ival = *idst + 1;  
*idst = ival;  
`interrupt enable`  
return (ival);
```

ATM\_inci is written in assembly language, efficiently disabling interrupts on the target processor during the call.

Incrementing a value equal to the maximum signed integer results in a value equal to the minimum signed integer.

**See Also**

ATM\_deci  
ATM\_incu

<b>ATM_incu</b>	<i>Atomically increment Uns memory and return new value</i>
<b>C Interface</b>	
<b>Syntax</b>	uval = ATM_incu(udst);
<b>Parameters</b>	volatile Uns *udst; /* pointer to unsigned */
<b>Return Value</b>	Uns uval; /* new value after increment */
<b>Assembly Interface</b>	none
<b>Description</b>	<p>ATM_incu atomically increments an Uns memory location and returns its new value as follows:</p> <pre>`interrupt disable` uval = *udst + 1; *udst = uval; `interrupt enable` return (uval);</pre> <p>ATM_incu is written in assembly language, efficiently disabling interrupts on the target processor during the call.</p> <p>Incrementing a value equal to the maximum unsigned integer results in a value equal to the minimum unsigned integer.</p>
<b>See Also</b>	ATM_decu ATM_inci

**ATM\_ori***Atomically OR Int memory location and return previous value***C Interface****Syntax**`ival = ATM_ori(idst, isrc);`**Parameters**

<code>volatile Int</code>	<code>*idst;</code>	<code>/* pointer to integer */</code>
<code>Int</code>	<code>isrc;</code>	<code>/* integer mask */</code>

**Return Value**

<code>Int</code>	<code>ival;</code>	<code>/* previous value of *idst */</code>
------------------	--------------------	--

**Assembly Interface**`none`**Description**

ATM\_ori atomically ORs the mask contained in isrc with a destination memory location and overwrites the destination value \*idst with the result as follows:

```
`interrupt disable`  
ival = *idst;  
*idst = ival | isrc;  
`interrupt enable`  
return(ival);
```

ATM\_ori is written in assembly language, efficiently disabling interrupts on the target processor during the call.

**See Also**

ATM\_andi  
ATM\_oru

ATM\_oru

Atomically OR Uns memory location and return previous value

C Interface

Syntax	uval = ATM_oru(udst, usrc);
Parameters	volatile Uns *udst; /* pointer to unsigned */ Uns usrc; /* unsigned mask */
Return Value	Uns uva; /* previous value of *udst */
Assembly Interface	none
Description	<p>ATM_oru atomically ORs the mask contained in usrc with a destination memory location and overwrites the destination value *udst with the result as follows:</p> <pre>`interrupt disable` uval = *udst; *udst = uval   usrc; `interrupt enable` return(uval);</pre> <p>ATM_oru is written in assembly language, efficiently disabling interrupts on the target processor during the call.</p>
See Also	ATM_andu ATM_ori

**ATM\_seti***Atomically set Int memory and return previous value***C Interface****Syntax**`iold = ATM_seti(idst, inew);`**Parameters**

<code>volatile Int</code>	<code>*idst;</code>	<code>/* pointer to integer */</code>
<code>Int</code>	<code>inew;</code>	<code>/* new integer value */</code>

**Return Value**

<code>Int</code>	<code>iold;</code>	<code>/* previous value of *idst */</code>
------------------	--------------------	--

**Assembly Interface**

none

**Description**

ATM\_seti atomically sets an Int memory location to a new value and returns its previous value as follows:

```
`interrupt disable`
ival = *idst;
*idst = inew;
`interrupt enable`
return (ival);
```

ATM\_seti is written in assembly language, efficiently disabling interrupts on the target processor during the call.

**See Also**

ATM\_setu  
ATM\_cleari

ATM\_setu

Atomically set Uns memory and return previous value

C Interface

Syntax	uold = ATM_setu(udst, unew);
Parameters	volatile Uns *udst; /* pointer to unsigned */ Uns unew; /* new unsigned value */
Return Value	Uns uold; /* previous value of *udst */

Assembly Interface

none

Description

ATM\_setu atomically sets an Uns memory location to a new value and returns its previous value as follows:

```
`interrupt disable`  
uval = *udst;  
*udst = unew;  
`interrupt enable`  
return (uval);
```

ATM\_setu is written in assembly language, efficiently disabling interrupts on the target processor during the call.

See Also

ATM\_clearu  
ATM\_seti

## 2.3 C54 and C55 Modules

### Functions



The C54 and C55 modules include target-specific functions for the TMS320C5000 family

- ❑ C54\_disableIMR. ASM macro to disable selected interrupts in the IMR
- ❑ C54\_enableIMR. ASM macro to enable selected interrupts in the IMR
- ❑ C54\_plug. Plug interrupt vector
- ❑ C55\_disableIER0, C55\_disableIER1. ASM macros to disable selected interrupts in the IER0/IER1, respectively
- ❑ C55\_enableIER0, C55\_enableIER1. ASM macros to enable selected interrupts in the IER0/IER1, respectively
- ❑ C55\_plug. Plug interrupt vector

### Description

The C54 and C54 modules provide certain target-specific functions and definitions for the TMS320C5000 family of processors.

See the c54.h or c54.h files for the complete list of definitions for hardware flags for C. The c54.h and c54.h files contain C language macros, #defines for various TMS320C5000 registers, and structure definitions. The c54.h54 and c54.h54 files also contain assembly language macros for saving and restoring registers in interrupt service routines.



<b>C54_disableIMR</b>	<i>Disable certain maskable interrupts</i>
<b>C Interface</b>	
<b>Syntax</b>	oldmask = C54_disableIMR(mask);
<b>Parameters</b>	Uns            mask;     /* disable mask */
<b>Return Value</b>	Uns            oldmask; /* actual bits cleared by disable mask */
<b>Assembly Interface</b>	none
<b>Description</b>	<p>C54_disableIMR disables interrupts by clearing the bits specified by mask in the Interrupt Mask Register (IMR).</p> <p>The C version of C54_disableIMR returns a mask of bits actually cleared. This return value should be passed to C54_enableIMR to re-enable interrupts.</p> <p>See C54_enableIMR for a description and code examples for safely protecting a critical section of code from interrupts.</p>
<b>See Also</b>	C54_enableIMR

**C55\_disableIER0,  
C55\_disableIER1***Disable certain maskable interrupts***C Interface**

<b>Syntax</b>	oldmask = C55_disableIER0(mask); oldmask = C55_disableIER1(mask);
<b>Parameters</b>	Uns            mask;     /* disable mask */
<b>Return Value</b>	Uns            oldmask; /* actual bits cleared by disable mask */

**Assembly Interface**

<b>Syntax</b>	C55_disableIER0 IEMASK, REG0 C55_disableIER1 IEMASK, REG0
<b>Preconditions</b>	IEMASK        ; interrupt disable mask REG0           ; temporary register that can be modified
<b>Postconditions</b>	none

**Description**            C55\_disableIER0 and C55\_disableIER1 disable interrupts by clearing the bits specified by mask in the Interrupt Enable Register (IER0/IER1).

The C versions of C55\_disableIER0 and C55\_disableIER1 return a mask of bits actually cleared. This return value should be passed to C55\_enableIER0 or C55\_enableIER1 to re-enable interrupts.

See C55\_enableIER0 and C55\_enableIER1 for a description and code examples for safely protecting a critical section of code from interrupts.

**See Also**                C55\_enableIER0 and C55\_enableIER1

C54\_enableIMR

Enable certain maskable interrupts

C Interface

Syntax	C54_enableIMR(oldmask);
Parameters	Uns            oldmask; /* enable mask */
Return Value	Void

Assembly Interface            none

**Description**            C54\_disableIMR and C54\_enableIMR are used to disable and enable specific internal interrupts by modifying the Interrupt Mask Register (IMR). C54\_disableIMR clears the bits specified by the mask parameter in the IMR and returns a mask of the bits it cleared. C54\_enableIMR sets the bits specified by the oldmask parameter in the IMR.

C54\_disableIMR and C54\_enableIMR are usually used in tandem to protect a critical section of code from interrupts. The following code example shows a region protected from all interrupts:

```
/* C example */
Uns  oldmask;

oldmask = C54_disableIER(~0);
    `do some critical operation; `
    `do not call TSK_sleep, SEM_post, etc.`
C54_enableIER(oldmask);
```

---

**Note:**

DSP/BIOS kernel calls that can cause a task switch (for example, SEM\_post and TSK\_sleep) should be avoided within a C54\_disableIMR / C54\_enableIMR block since the interrupts can be disabled for an indeterminate amount of time if a task switch occurs.

---

Alternatively, you can disable DSP/BIOS task scheduling for this block by enclosing it with TSK\_disable and TSK\_enable. You can also use C54\_disableIMR / C54\_enableIMR to disable selected interrupts, allowing other interrupts to occur. However, if another HWI does occur during this region, it could cause a task switch. You can prevent this by using TSK\_disable / TSK\_enable around the entire region:

```
Uns    oldmask;

TSK_disable();
oldmask = C54_disableIMR(INTMASK);
    `do some critical operation;`
    `NOT OK to call TSK_sleep, SEM_post, etc.`
C54_enableIMR(oldmask);
TSK_enable();
```

---

**Note:**

If you use C54\_disableIMR and C54\_enableIMR to disable only some interrupts, you must surround this region with SWI\_disable and SWI\_enable, to prevent an intervening HWI from causing an SWI or TSK switch.

---

The second approach is preferable if it is important not to disable all interrupts in your system during the critical operation.

**See Also**

C54\_disableIMR

**C55\_enableIER0,  
C55\_enableIER1***Enable certain maskable interrupts***C Interface**

<b>Syntax</b>	C55_enableIER0(oldmask); C55_enableIER1(oldmask);
<b>Parameters</b>	Uns            oldmask; /* enable mask */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	C55_enableIER0 IEMASK, REG0 C55_enableIER1 IEMASK, REG0
<b>Preconditions</b>	IEMASK        ; interrupt enable mask REG0          ; temporary register that can be modified by the API
<b>Postconditions</b>	none

**Description**

C55\_disableIER0 and C55\_disableIER1 and C55\_enableIER0 and C55\_enableIER1 disable and enable specific internal interrupts by modifying the Interrupt Enable Register (IER0/IER1). C55\_disableIER0 and C55\_disableIER1 clear the bits specified by the mask parameter in the Interrupt Mask Register and return a mask of the bits it cleared. C55\_enableIER0 and C55\_enableIER1 set the bits specified by the oldmask parameter in the Interrupt Mask Register.

C55\_disableIER0 and C55\_disableIER1 and C55\_enableIER0 and C55\_enableIER1 are usually used in tandem to protect a critical section of code from interrupts. The following code examples show a region protected from all maskable interrupts:

```
; ASM example

.include c55.h55
...

; disable interrupts specified by IEMASK
C55_disableIER0 IEMASK0, t0
C55_disableIER1 IEMASK1, t1

`do some critical operation`

; enable interrupts specified by IEMASK
C55_enableIER0 IEMASK0, t0
C55_enableIER1 IEMASK1, t1
```

```

/* C example */
Uns    oldmask;

oldmask0 = C55_disableIER0(~0);
`do some critical operation; `
`do not call TSK_sleep, SEM_post, etc.`
C55_enableIER0(oldmask0);

```

---

**Note:**

DSP/BIOS kernel calls that can cause rescheduling of tasks (for example, SEM\_post and TSK\_sleep) should be avoided within a C55\_disableIER0 / C55\_disableIER1 and C55\_enableIER0 / C55\_enableIER1 block since the interrupts can be disabled for an indeterminate amount of time if a task switch occurs.

---

You can use C55\_disableIER0 / C55\_disableIER1 and C55\_enableIER0 / C55\_enableIER1 to disable selected interrupts, while allowing other interrupts to occur. However, if another ISR occurs during this region, it could cause a task switch. You can prevent this by enclosing it with TSK\_disable / TSK\_enable. to disable DSP/BIOS task scheduling.

```

Uns    oldmask;

TSK_disable();
oldmask0 = C55_disableIER0(INTMASK0);
oldmask1 = C55_disableIER1(INTMASK1);
`do some critical operation; `
`NOT OK to call TSK_sleep, SEM_post, etc.`
C55_enableIER0(oldmask0);
C55_enableIER0(oldmask1);
TSK_enable();

```

---

**Note:**

If you use C55\_disableIER0 / C55\_disableIER1 and C55\_enableIER0 / C55\_enableIER1 to disable only some interrupts, you must surround this region with SWI\_disable / SWI\_enable, to prevent an intervening HWI from causing a SWI or TSK switch.

---

The second approach is preferable if it is important not to disable all interrupts in your system during the critical operation.

**See Also**

C55\_disableIMR

**C54\_plug**

*C function to plug an interrupt vector*

**C Interface**

<b>Syntax</b>	C54_plug(vecid, fxn);
<b>Parameters</b>	Int            vecid;    /* interrupt id */ Fxn           fxn;       /* pointer to HWI function */
<b>Return Value</b>	Void
<b>Assembly Interface</b>	none
<b>Description</b>	<p>C54_plug writes a branch vector into the interrupt vector table, at the address corresponding to vecid. The op-codes written in the branch vector create a branch to the function entry point specified by fxn:</p> <p>b            fxn</p> <p>C54_plug does not enable the interrupt. Use C54_enableIMR to enable specific interrupts.</p>
<b>Constraints and Calling Context</b>	<p>❑ vecid must be a valid interrupt ID in the range of 0-31.</p>
<b>See Also</b>	C54_enableIMR

**C55\_plug***C function to plug an interrupt vector***C Interface****Syntax**`C55_plug(vecid, fxn);`**Parameters**

Int	vecid;	/* interrupt id */
Fxn	fxn;	/* pointer to HWI function */

**Return Value**

Void

**Assembly Interface**

none

**Description**

C55\_plug hooks up the specified function as the branch target or a hardware interrupt (fielded by the CPU) at the vector address specified in vecid. C55\_plug does not enable the interrupt. Use or C55\_enableIER0 and C55\_enableIER1 to enable specific interrupts.

**Constraints and  
Calling Context**

- ❑ vecid must be a valid interrupt ID in the range of 0-31.

**See Also**

C55\_enableIER



## 2.4 CLK Module

The CLK module is the system clock manager.

### Functions

- ☐ `CLK_countspms`. Timer counts per millisecond
- ☐ `CLK_gethtime`. Get high resolution time
- ☐ `CLK_gettime`. Get low resolution time
- ☐ `CLK_getprd`. Get period register value

### Description

The CLK module provides a method for invoking functions periodically.

DSP/BIOS provides two separate timing methods: the high- and low-resolution times managed by the CLK module and the system clock. In the default configuration, the low-resolution time and the system clock are the same.

The CLK module provides a real-time clock with functions to access this clock at two resolutions. This clock can be used to measure the passage of time in conjunction with STS accumulator objects, as well as to add timestamp messages to event logs. Both the low-resolution and high-resolution times are stored as 32-bit values. The value restarts at the value in the period register when 0 is reached.

If the Clock Manager is enabled in the Configuration Tool, the time counter is decremented at the following rate, where CLKOUT is the DSP clock speed in MHz (see the Global Settings Property dialog) and TDDR is the value of the timer divide-down register (see the CLK Manager Property dialog):

$$\text{CLKOUT} / (\text{TDDR} + 1)$$

When this register reaches 0, the counter is reset to the value in the period register and a timer interrupt occurs. When a timer interrupt occurs, the HWI object for the 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 HWI

Therefore, the low-resolution clock ticks at the timer interrupt rate and the clock's value is equal to the number of timer interrupts that have occurred. You can use the `CLK_gettime` function to get the low-resolution time and the `CLK_getprd` function to get the value of the period register property.

The high-resolution time is the number of times the timer counter register has been decremented (number of instruction cycles). Given the high

CPU clock rate, the 16-bit timer counter register wraps around quite fast. The 32-bit high-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding the difference between the value in the period register and the current value of the timer counter register. You can use the `CLK_gettime` function to get the high-resolution time and the `CLK_countspms` function to get the number of hardware timer counter register ticks per millisecond.

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 can only invoke DSP/BIOS calls that are allowable from within an HWI.

---

**Note:**

CLK functions should not call `HWI_enter` and `HWI_exit` as these are called internally by DSP/BIOS when it runs `CLK_F_isr`. Additionally, CLK functions should **not** use the `interrupt` keyword or the `INTERRUPT` pragma in C functions.

---

If you do not want the on-device timer to drive the system clock, you can disable the CLK Manager by clearing the Enable CLK Manager checkbox on the CLK Manager Properties dialog. If this box is gray, go to the PRD Manager Properties dialog and clear the Use CLK Manager to Drive PRD box. Then you can disable the CLK Manager.

## Clock Manager Properties

The following global properties can be set for the CLK module on the CLK Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the CLK objects created with the Configuration Tool.
- ☐ **Enable CLK Manager.** If checked, the on-device timer hardware is used to drive the high- and low-resolution times and to trigger execution of CLK functions.
- ☐ **Use high resolution time for internal timings.** If checked, the high-resolution timer is used to monitor internal periods; otherwise the less intrusive, low-resolution timer is used.
- ☐ **Microseconds/Int.** The number of microseconds between timer interrupts. The period register is set to a value that achieves the desired period as closely as possible.
- ☐ **Directly configure on-device timer registers.** If checked, the timer's hardware registers, PRD and TDDR, can be directly set to the

desired values. In this case, the Microseconds/Int field is computed based on the values in PRD and TDDR and the CPU clock speed.

- ☐ **Fix TDDR.** If checked, the value in the TDDR field is not modified by changes to the Microseconds/Int field.
- ☐ **TDDR Register.** The on-device timer divide-down register.
- ☐ **PRD Register.** The on-device timer period register.
- ☐ **Instructions/Int.** The number of instruction cycles represented by the period specified above. This is an informational field only.

## CLK Object Properties

The Clock Manager allows you to create an arbitrary number of clock functions. Clock functions are functions executed by the Clock Manager every time a timer interrupt occurs. These functions can invoke any DSP/BIOS operations allowable from within an HWI except HWI\_enter or HWI\_exit.

The following properties can be set for a clock function object on the CLK Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this CLK object.
- ☐ **function.** The function to be executed when the timer hardware interrupt occurs. This function must be written like an HWI function; it must be written in C or assembly and must save and restore any registers this function modifies. However, this function can not call HWI\_enter or HWI\_exit because DSP/BIOS calls them internally before and after this function runs.

These functions should be very short as they are performed frequently.

Since all CLK functions are performed at the same periodic rate, functions that need to run at a multiple of that rate should either count the number of interrupts and perform their activities when the counter reaches the appropriate value or be configured as PRD objects.

If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code, which must use leading underscores when referencing C functions or labels.)

## CLK - Code Composer Studio Interface

To enable CLK logging, choose DSP/BIOS→RTA Control Panel and put a check in the appropriate box. You see indicators for low resolution clock interrupts in the Time row of the Execution Graph, which you can open by choosing DSP/BIOS→Execution Graph.

**CLK\_countspms***Number of hardware timer counts per millisecond***C Interface****Syntax** `ncounts = CLK_countspms();`**Parameters** Void**Return Value** LgUns ncounts;**Assembly Interface****Syntax** CLK\_countspms**Preconditions** none**Postconditions** a = the number of hardware timer register ticks per millisecond**Modifies** ag, ah, al, c**Assembly Interface****Syntax** CLK\_countspms**Preconditions** none**Postconditions** ac0 = the number of hardware timer register ticks per millisecond**Modifies** ac0g, ac0h, ac0l**Reentrant** yes

**Description** CLK\_countspms returns the number of hardware timer register ticks per millisecond. This corresponds to the number of high-resolution ticks per millisecond.

CLK\_countspms can be used to compute an absolute length of time from the number of hardware timer interrupts. For example, the following returns the number of milliseconds since the 32-bit high-resolution time last wrapped back to the value in the period register:

```
timeAbs = (CLK_gettime() * (CLK_getprd() + 1)) /  
CLK_countspms();
```

**See Also**

CLK\_gettime

CLK\_getprd

STS\_delta

**CLK\_geththime***Get high-resolution time***C Interface**

<b>Syntax</b>	<code>currtime = CLK_geththime();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	LgUns <code>currtime</code> <i>/* high-resolution time */</i>

**Assembly Interface**

<b>Syntax</b>	<code>CLK_geththime</code>
<b>Preconditions</b>	<code>intm = 1</code> <code>cpl = ovm = c16 = frct = cmpt = 0</code>
<b>Postconditions</b>	<code>ah</code> = bits 32 - 16 of high-resolution time <code>al</code> = bits 15 - 0 of high-resolution time
<b>Modifies</b>	<code>ag</code> , <code>ah</code> , <code>al</code> , <code>ar5</code> , <code>bg</code> , <code>bh</code> , <code>bl</code> , <code>c</code> , <code>dp</code> , <code>t</code> , <code>tc</code>

**Assembly Interface**

<b>Syntax</b>	<code>CLK_geththime</code>
<b>Preconditions</b>	<code>intm = 1</code>
<b>Postconditions</b>	<code>ac0h</code> = bits 32 - 16 of high-resolution time <code>ac0l</code> = bits 15 - 0 of high-resolution time
<b>Modifies</b>	<code>ac0g</code> , <code>ac0h</code> , <code>ac0l</code> , <code>ac1g</code> , <code>ac1h</code> , <code>ac1l</code> , <code>t0</code> , <code>t1</code>

**Reentrant**      no

**Description**      CLK\_geththime returns the number of high resolution clock cycles that have occurred as a 32-bit time value. When the number of cycles reaches the maximum value that can be stored in 32 bits, the value wraps back to 0.

High-resolution time is the number of times the timer counter register has been decremented. When the CLK manager is enabled in the

Configuration Tool, the time counter is decremented at the following rate, where CLKOUT is the DSP clock speed in MHz (see the Global Settings Property dialog) and TDDR is the value of the timer divide-down register (see the CLK Manager Property dialog):

$$\text{CLKOUT} / (\text{TDDR} + 1)$$

When this register reaches 0, the counter is reset to the value in the period register and a timer interrupt occurs. When a timer interrupt occurs, the HWI object for the timer runs the CLK\_F\_isr function.

In contrast, CLK\_gethtime returns the number of timer interrupts that have occurred. When the timer counter register reaches 0, the counter is reset to the value set for the period register property of the CLK module and a timer interrupt occurs.

High-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding to it the difference between the period and the timer register values. Although the CLK\_gethtime uses the period register value to calculate the high-resolution time, the value of the high-resolution time is independent of the actual value in the period register. This is because the timer counter register is divided by the period register value when incrementing the low-resolution time, and the result is multiplied by the same period register value to calculate the high-resolution time.

CLK\_gethtime provides a value with greater accuracy than CLK\_gettime, but which wraps back to 0 more frequently. For example, if the device's clock rate is 200 MHz, then regardless of the period register value, the CLK\_gethtime value wraps back to 0 approximately every 86 seconds.

CLK\_gethtime can be used in conjunction with STS\_set and STS\_delta to benchmark code. CLK\_gethtime can also be used to add a time stamp to event logs.

- ❑ CLK\_gethtime cannot be called from the program's main function.

## Constraints and Calling Context

### Example

```
/* ===== showTime ===== */  
  
Void showTicks  
{  
    LOG_printf(&trace, "time = %d", CLK_gethtime());  
}
```

### See Also

CLK\_gettime  
PRD\_getticks  
STS\_delta

**CLK\_gettime***Get low-resolution time***C Interface**

<b>Syntax</b>	<code>currtime = CLK_gettime();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	LgUns      currtime    /* low-resolution time */

**Assembly Interface**

<b>Syntax</b>	CLK_gettime
<b>Preconditions</b>	none
<b>Postconditions</b>	ah = bits 32 - 16 of low-resolution time al = bits 15 - 0 of low-resolution time
<b>Modifies</b>	ag, ah, al, c

**Assembly Interface**

<b>Syntax</b>	CLK_gettime
<b>Preconditions</b>	none
<b>Postconditions</b>	ac0h = bits 32 - 16 of low-resolution time ac0l = bits 15 - 0 of low-resolution time
<b>Modifies</b>	ac0g, ac0h, ac0l

<b>Reentrant</b>	yes
------------------	-----

<b>Description</b>	CLK_gettime returns the number of timer interrupts that have occurred as a 32-bit time value. When the number of interrupts reaches the maximum value that can be stored in 32 bits, value wraps back to 0 on the next interrupt.
--------------------	---

The low-resolution time is the number of timer interrupts that have occurred.



The timer counter is decremented every instruction cycle. When this register reaches 0, the counter is reset to the value set for the period register property of the CLK module and a timer interrupt occurs. When a timer interrupt occurs, all the functions specified by CLK objects are performed in sequence in the context of that HWI.

The default low resolution interrupt rate is 1 millisecond/interrupt. By adjusting the period register, you can set rates from less than 1 microsecond/interrupt to more than 1 second/interrupt.

If you use the default configuration, the system clock rate matches the low-resolution rate.

In contrast, CLK\_gettime returns the number of high resolution clock cycles that have occurred. When the timer counter register reaches 0, the counter is reset to the value set for the period register property of the CLK module and a timer interrupt occurs.

Therefore, CLK\_gettime provides a value with greater accuracy than CLK\_gettime, but which wraps back to 0 more frequently. For example, if the device's clock rate is 80 MHz, and you use the default period register value of 40000, the CLK\_gettime value wraps back to 0 approximately every 107 seconds, while the CLK\_gettime value wraps back to 0 approximately every 49.7 days.

CLK\_gettime is often used to add a time stamp to event logs for events that occur over a relatively long period of time.

- ❑ CLK\_gettime cannot be called from the program's main function.

## Constraints and Calling Context

### Example

```
/* ===== showTicks ===== */

Void showTicks
{
    LOG_printf(&trace, "time = 0x%x %x",
        (Int)(CLK_gettime() >> 16), (Int)CLK_gettime());
}
```

### See Also

CLK\_gettime  
PRD\_getticks  
STS\_delta

**CLK\_getprd***Get period register value***C Interface**

<b>Syntax</b>	period = CLK_getprd();
<b>Parameters</b>	Void
<b>Return Value</b>	Uns            period    /* period register value */

**Assembly Interface**

<b>Syntax</b>	CLK_getprd
<b>Preconditions</b>	none
<b>Postconditions</b>	a = the value set for the period register property
<b>Modifies</b>	ag, ah, al, c

**Assembly Interface**

<b>Syntax</b>	CLK_getprd
<b>Preconditions</b>	none
<b>Postconditions</b>	ac0 = the value set for the period register property
<b>Modifies</b>	ac0g, ac0h, ac0l

<b>Reentrant</b>	yes
------------------	-----

<b>Description</b>	CLK_getprd returns the value set for the period register property of the CLK Manager in the Configuration Tool. CLK_getprd can be used to compute an absolute length of time from the number of hardware timer interrupts. For example, the following code returns the number of milliseconds since the 32-bit low-resolution time last wrapped back to 0:
--------------------	--

```
timeAbs = (CLK_getltime() * (CLK_getprd() + 1)) /
CLK_countspms();
```

<b>See Also</b>	CLK_countspms CLK_gethtime STS_delta
-----------------	--

## 2.5 DEV Module

The DEV module is the device driver interface.

### Functions

- ❑ DEV\_match. Match device name with driver
- ❑ Dxx\_close. Close device
- ❑ Dxx\_ctrl. Device control
- ❑ Dxx\_idle. Idle device
- ❑ Dxx\_init. Initialize device
- ❑ Dxx\_issue. Send frame to device
- ❑ Dxx\_open. Open device
- ❑ Dxx\_ready. Device ready
- ❑ Dxx\_reclaim. Retrieve frame from device

### Constants, Types, and Structures

```
#define DEV_INPUT      0
#define DEV_OUTPUT     1

typedef struct DEV_Frame { /* frame object */
    QUE_Elem    link;      /* queue link */
    Ptr         addr;      /* buffer address */
    Uns         size;      /* buffer size */
    Arg         misc;      /* reserved for driver */
    Arg         arg;       /* user argument */
} DEV_Frame;

typedef struct DEV_Obj { /* device object */
    QUE_Handle  todevice;  /* downstream frames here */
    QUE_Handle  fromdevice; /* upstream frames here */
    Uns         bufsize;   /* buffer size */
    Uns         nbufs;     /* number of buffers */
    Int         segid;     /* buffer segment ID */
    Int         mode;      /* DEV_INPUT/DEV_OUTPUT */
    Int         devid;     /* device ID */
#ifdef (_54_) && defined(_FAR_MODE)
    || defined(_55_)
    LgInt       devid;    /* device ID */
#else
    Int         devid;    /*device ID */
#endif
    Ptrparams;           /* device parameters */
    Ptr         object;   /* ptr to device instance obj */
    DEV_Fxns    fxns;     /* driver functions */
    Uns         timeout;  /* SIO_reclaim timeout value */
    Uns         align;    /* buffer alignment */
} DEV_Obj;
```

```

typedef struct DEV_Fxns { /* driver function table */
    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;

typedef struct DEV_Device { /* device specifier */
    String     name;          /* device name */
    DEV_Fxns   *fxns;         /* device function table*/
    Int        devid;         /* device ID */
#ifdef _54_ && defined(_FAR_MODE)
    || defined(_55_)
    LgInt      devid;         /* device ID */
#else
    Int devid; /*device ID */
#endif

    Ptr        params;        /* device parameters */
} DEV_Device;

```

## Description

Using generic functions provided by the SIO module, programs indirectly invoke corresponding functions which manage the particular device attached to the stream. Unlike other modules, your application programs do not issue direct calls to driver functions that manipulate individual device objects managed by the module. Instead, each driver module exports a distinguished structure of type `DEV_Fxns`, which is used by the SIO module to route generic function calls to the proper driver function.

The Dxx functions are templates for driver functions. To ensure that all driver functions present an identical interface to DEV, the driver functions must follow these templates.

## DEV Manager Properties

The default configuration contains managers for the following built-in device drivers:

- ❑ DGN software generator driver. A pseudo-device that generates one of several data streams, such as a sin/cos series or white noise. This driver can be useful for testing applications that require an input stream of data.
- ❑ DHL host link driver. A driver that uses the HST interface to send data to and from the DSP/BIOS Host Channel Control Analysis Tool.
- ❑ DPI pipe driver. A software device used to stream data between DSP/BIOS tasks.

To configure devices for other drivers, use the Configuration Tool to insert a User-defined Device object. There are no global properties for the user-defined device manager.

The following additional device drivers are supplied with DSP/BIOS:

- ☐ DAX. Generic streaming device driver
- ☐ DGS. Stackable gather/scatter driver
- ☐ DNL. Null driver
- ☐ DOV. Stackable overlap driver
- ☐ DST. Stackable “split” driver
- ☐ DTR. Stackable streaming transformer driver

## DEV Object Properties

The following properties can be set on the DEV Object Properties dialog in the Configuration Tool for a user-defined device:

- ☐ **comment.** Type a comment to identify this object.
- ☐ **DEV\_Fxns table.** Specify the name of the device functions table contained in `dx.c`. This table should have a name with the format `DXX_FXNS` where `XX` is the two-letter code for the driver used by this device.

Use a leading underscore before the table name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)

- ☐ **Parameters.** If this device uses additional parameters, provide the name of the parameter structure. This structure should have a name with the format `DXX_Params` where `XX` is the two-letter code for the driver used by this device.

Use a leading underscore before the structure name.

- ☐ **Device ID.** Specify the device ID. If the value you provide is non-zero, the value takes the place of a value that would be appended to the device name in a call to `SIO_create`. The purpose of such a value is driver-specific.
- ☐ **Init Fxn.** Specify the function to run to initialize this device. Use a leading underscore before the function name if the function is written in C.
- ☐ **Stacking Device.** Put a checkmark in this box if device uses a stacking driver.

**DEV\_match***Match a device name with a driver***C Interface**

**Syntax**                      substr = DEV\_match(name, device);

**Parameters**                String            name;     /\* device name \*/  
DEV\_Device \*\*device; /\* pointer to device table entry \*/

**Return Value**              String            substr;     /\* remaining characters after match \*/

**Assembly Interface**        none

**Description**                DEV\_match searches the device table for the first device name that matches a prefix of name. The output parameter, device, points to the appropriate entry in the device table if successful and is set to NULL on error.

A pointer to the characters remaining after the match is returned in substr. This string is used by stacking devices to specify the name(s) of underlying devices (for example, /scale10/sine might match /scale10 a stacking device which would, in turn, use /sine to open the underlying generator device).

**See Also**                    SIO\_create

**Dxx\_close***Close device***C Interface**

<b>Syntax</b>	<code>status = Dxx_close(device);</code>
<b>Parameters</b>	<code>DEV_Handle device; /* device handle */</code>
<b>Return Value</b>	<code>Int status; /* result of operation */</code>

**Assembly Interface** none

**Description** Dxx\_close closes the device associated with device and returns an error code indicating success (SYS\_OK) or failure. device is bound to the device through a prior call to Dxx\_open.

SIO\_delete first calls Dxx\_idle to idle the device. Then it calls Dxx\_close.

Once device has been closed, the underlying device is no longer accessible via this descriptor.

**Constraints and Calling Context**

❑ device must be bound to a device by a prior call to Dxx\_open.

**See Also**

Dxx\_idle  
Dxx\_open  
SIO\_delete

Dxx_ctrl	Device control operation
C Interface	
Syntax	status = Dxx_ctrl(device, cmd, arg);
Parameters	DEV_Handle device    /* device handle */ Uns                    cmd;    /* driver control code */ Arg                    arg;    /* control operation argument */
Return Value	Int                    status;    /* result of operation */
Assembly Interface	none
Description	<p>Dxx_ctrl performs a control operation on the device associated with device and returns an error code indicating success (SYS_OK) or failure. The actual control operation is designated through cmd and arg, which are interpreted in a driver-dependent manner.</p> <p>Dxx_ctrl is called by SIO_ctrl to send control commands to a device.</p>
Constraints and Calling Context	<div> <div></div> <div>device must be bound to a device by a prior call to Dxx_open.</div> </div>
See Also	SIO_ctrl



Dxx\_idle

Idle device

C Interface

Syntax	status = Dxx_idle(device, flush);
Parameters	DEV_Handle device; /* device handle */ Bool flush; /* flush output flag */
Return Value	Int status; /* result of operation */
Assembly Interface	none
Description	<p>Dxx_idle places the device associated with device into its idle state and returns an error code indicating success (SYS_OK) or failure. Devices are initially in this state after they are opened with Dxx_open.</p> <p>Dxx_idle is called by SIO_idle, SIO_flush, and SIO_delete to recycle frames to the appropriate queue.</p> <p>flush is a boolean parameter that indicates what to do with any pending data of an output stream while returning the device to its initial state. If flush is TRUE, all pending data is discarded and Dxx_idle does not block waiting for data to be processed. If flush is FALSE, the Dxx_idle function does not return until all pending output data has been rendered. All pending data in an input stream is always discarded, without waiting.</p>
Constraints and Calling Context	<div><input type="checkbox"/> device must be bound to a device by a prior call to Dxx_open.</div>
See Also	SIO_delete SIO_idle SIO_flush

---

<b>Dxx_init</b>	<i>Initialize device</i>
-----------------	--------------------------

**C Interface**

<b>Syntax</b>	Dxx_init();
<b>Parameters</b>	Void
<b>Return Value</b>	Void

**Assembly Interface**      none

**Description**      Dxx\_init is used to initialize the device driver module for a particular device. This initialization often includes resetting the actual device to its initial state.

Dxx\_init is called at system startup, before the application's main function is called.

**Dxx\_issue***Send a buffer to the device***C Interface**

<b>Syntax</b>	<code>status = Dxx_issue(device);</code>
<b>Parameters</b>	<code>DEV_Handle device;    /* device handle */</code>
<b>Return Value</b>	<code>Int                    status;    /* result of operation */</code>

**Assembly Interface**

none

**Description**

Dxx\_issue is used to notify a device that a new frame has been placed on the device->todevice queue. If the device was opened in DEV\_INPUT mode then Dxx\_issue uses this frame for input. If the device was opened in DEV\_OUTPUT mode, Dxx\_issue processes the data in the frame, then outputs it. In either mode, Dxx\_issue ensures that the device has been started, and returns an error code indicating success (SYS\_OK) or failure.

Dxx\_issue does not block. In output mode it processes the buffer and places it in a queue to be rendered. In input mode, it places a buffer in a queue to be filled with data, then returns.

Dxx\_issue is used in conjunction with Dxx\_reclaim to operate a stream. The Dxx\_issue call sends a buffer to a stream, and the Dxx\_reclaim retrieves a buffer from a stream. Dxx\_issue performs processing for output streams, and provides empty frames for input streams. The Dxx\_reclaim recovers empty frames in output streams, retrieves full frames, and performs processing for input streams.

SIO\_issue calls Dxx\_issue after placing a new input frame on the device->todevice. If Dxx\_issue fails, it should return an error code. Before attempting further I/O through the device, the device should be idled, and all pending buffers should be flushed if the device was opened for DEV\_OUTPUT.

In a stacking device, Dxx\_issue must preserve all information in the DEV\_Frame object except link and misc. On a device opened for DEV\_INPUT, Dxx\_issue should preserve the size and the arg fields. On a device opened for DEV\_OUTPUT, Dxx\_issue should preserve the buffer data (transformed as necessary), the size (adjusted as appropriate by the transform) and the arg field. The DEV\_Frame objects themselves do not need to be preserved, only the information they contain.

Dxx\_issue must preserve and maintain buffers sent to the device so they can be returned in the order they were received, by a call to Dxx\_reclaim.

**Constraints and  
Calling Context**

- ❑ device must be bound to a device by a prior call to `Dxx_open`.

**See Also**

`Dxx_reclaim`  
`SIO_issue`  
`SIO_get`  
`SIO_put`

**Dxx\_open**

*Open device*

**C Interface**

<b>Syntax</b>	status = Dxx_open(device, name);
<b>Parameters</b>	DEV_Handle device; /* driver handle */ String name; /* device name */
<b>Return Value</b>	Int status; /* result of operation */

**Assembly Interface** none

**Description** Dxx\_open is called by SIO\_create to open a device. Dxx\_open opens a device and returns an error code indicating success (SYS\_OK) or failure.

The device parameter points to a DEV\_Obj whose fields have been initialized by the calling function (that is, SIO\_create). These fields can be referenced by Dxx\_open to initialize various device parameters. Dxx\_open is often used to attach a device-specific object to device->object. This object typically contains driver-specific fields that can be referenced in subsequent Dxx driver calls.

name is the string remaining after the device name has been matched by SIO\_create using DEV\_match.

**See Also** Dxx\_close  
SIO\_create

**Dxx\_ready***Check if device is ready for I/O***C Interface**

<b>Syntax</b>	<code>status = Dxx_ready(device, sem);</code>
<b>Parameters</b>	<code>DEV_Handle device;    /* device handle */</code> <code>SEM_Handle sem;       /* semaphore to post when ready */</code>
<b>Return Value</b>	<code>Bool            status;    /* TRUE if device is ready */</code>

**Assembly Interface**

none

**Description**

Dxx\_ready is called by SIO\_select to determine if the device is ready for an I/O operation. In this context, ready means a call that retrieves a buffer from a device does not block. If a frame exists, Dxx\_ready returns TRUE, indicating that the next SIO\_get, SIO\_put, or SIO\_reclaim operation on the device does not cause the calling task to block. If there are no frames available, Dxx\_ready returns FALSE. This informs the calling task that a call to SIO\_get, SIO\_put, or SIO\_reclaim for that device would result in blocking.

If the device is an input device that is not started, Dxx\_ready starts the device.

Dxx\_ready registers the device's ready semaphore with the SIO\_select semaphore sem. In cases where SIO\_select calls Dxx\_ready for each of several devices, each device registers its own ready semaphore with the unique SIO\_select semaphore. The first device that becomes ready calls SEM\_post on the semaphore.

SIO\_select calls Dxx\_ready twice; the second time, sem = NULL. This results in each device's ready semaphore being set to NULL. This information is needed by the Dxx HWI that normally calls SEM\_post on the device's ready semaphore when I/O is completed; if the device ready semaphore is NULL, the semaphore should not be posted.

**See Also**

SIO\_select

**Dxx\_reclaim***Retrieve a buffer from a device***C Interface**

<b>Syntax</b>	<code>status = Dxx_reclaim(device);</code>
<b>Parameters</b>	<code>DEV_Handle device; /* device handle */</code>
<b>Return Value</b>	<code>Int status; /* result of operation */</code>

**Assembly Interface**

none

**Description**

Dxx\_reclaim is used to request a buffer back from a device. Dxx\_reclaim does not return until a buffer is available for the client in the device->fromdevice queue. If the device was opened in DEV\_INPUT mode then Dxx\_reclaim blocks until an input frame has been filled with the number of MADUs requested, then processes the data in the frame and place it on the device->fromdevice queue. If the device was opened in DEV\_OUTPUT mode, Dxx\_reclaim blocks until an output frame has been emptied, then place the frame on the device->fromdevice queue. In either mode, Dxx\_reclaim blocks until it has a frame to place on the device->fromdevice queue, or until the stream's timeout expires, and it returns an error code indicating success (SYS\_OK) or failure.

If device->timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If device->timeout is SYS\_FOREVER, the task remains suspended until a frame is available on the device's fromdevice queue. If timeout is 0, Dxx\_reclaim returns immediately.

If timeout expires before a buffer is available on the device's fromdevice queue, Dxx\_reclaim returns SYS\_ETIMEOUT. Otherwise Dxx\_reclaim returns SYS\_OK for success, or an error code.

If Dxx\_reclaim fails due to a time out or any other reason, it does not place a frame on the device->fromdevice queue.

Dxx\_reclaim is used in conjunction with Dxx\_issue to operate a stream. The Dxx\_issue call sends a buffer to a stream, and the Dxx\_reclaim retrieves a buffer from a stream. Dxx\_issue performs processing for output streams, and provides empty frames for input streams. The Dxx\_reclaim recovers empty frames in output streams, and retrieves full frames and performs processing for input streams.

SIO\_reclaim calls Dxx\_reclaim, then it gets the frame from the device->fromdevice queue.

In a stacking device, Dxx\_reclaim must preserve all information in the DEV\_Frame object except link and misc. On a device opened for DEV\_INPUT, Dxx\_reclaim should preserve the buffer data (transformed as necessary), the size (adjusted as appropriate by the transform), and the arg field. On a device opened for DEV\_OUTPUT, Dxx\_reclaim should preserve the size and the arg field. The DEV\_Frame objects themselves do not need to be preserved, only the information they contain.

Dxx\_reclaim must preserve buffers sent to the device. Dxx\_reclaim should never return a buffer that was not received from the client through the Dxx\_issue call. Dxx\_reclaim always preserves the ordering of the buffers sent to the device, and returns with the oldest buffer that was issued to the device.

### **Constraints and Calling Context**

- ❑ device must be bound to a device by a prior call to Dxx\_open.

### **See Also**

Dxx\_issue  
SIO\_issue  
SIO\_get  
SIO\_put



**DAX Driver***Generic streaming device***Description**

The DAX driver works with most interrupt-driven A/D D/A devices or serial port devices. It requires a “controller” module for each device. The controller module consists of external functions bind, start, stop, and unbind, which DAX calls, in that order.

Individual controllers are configured for use by DAX via the DAX\_Params parameter structure and the Configuration Tool. The bind, start, stop, and unbind parameters are required, while the ctrl, outputdelay, and arg parameters are optional.

**Configuring a DAX Device**

To add a DAX device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table:** Type \_DAX\_FXNS
- ☐ **Parameters:** Type 0 (zero) to use the default parameters. To use different values, you must declare a DAX\_Params structure (as described after this list) containing the values to use for the parameters.
- ☐ **Device ID:** Type 0 (zero).
- ☐ **Init Fxn:** Type 0 (zero).
- ☐ **Stacking Device:** Do not put a checkmark in this box.

**DAX\_Params Elements**

DAX\_Params is defined in dax.h as follows:

```
/* ===== DAX_Params ===== */
typedef struct DAX_Params {
    Fxn    ctrl;    /* controller ctrl function */
    Fxn    stop;    /* controller stop function */
    Fxn    unbind;  /* controller unbind function */
    Arg    arg;     /* controller argument */
    Int    outputdelay; /* buffer delay before output */
} DAX_Params;
```

- ❑ **bind** is a controller function that initializes controller data structures and the actual device. The default is an empty function that returns 0. The bind function has the following parameters:

```
Cxx_bind(DAX_Handle port, String name);
```

name is a character string that can be used to specify additional parameters (for example, sample rate) for the controller.

For example, an A/D device might be configured with the name a2d. The application could create a stream with the Configuration Tool that uses the a2d device and the string value 32 in the Device Control Parameter property. The string 32 is passed by the DAX driver to Cxx\_bind, that presumably would parse 32 and initialize the device for a 32 kHz sample rate.

Alternatively, if the stream were created dynamically with SIO\_create, the application code would make the following call to cause the same effect:

```
stream = SIO_create("/a2d:32", SIO_INPUT, 128,
NULL);
```

- ❑ **ctrl** is a controller function that is used to send control commands to the device (for example to change the sample rate). The default is an empty function that returns 0. The ctrl function has the following parameters:

```
Cxx_ctrl(DEV_Handle device, Uns cmd, Arg arg);
```

- ❑ **start** is a controller function that starts the flow of data to or from the device. The default is an empty function that returns 0. The start function has the following parameters:

```
Cxx_start(DAX_Handle port);
```

- ❑ **stop** is a controller function that stops the flow of data to or from the device. The stop function has the following parameters:

```
Cxx_stop(DAX_Handle port);
```

- ❑ **unbind** is a controller function that resets controller data structures and frees any memory allocated by the bind function. The default is an empty function that returns 0. The unbind function has the following parameters:

```
Cxx_unbind(DAX_Handle port);
```

- ❑ **arg** is an optional parameter which can be used by the controller for configuration parameters specific for that controller. The default is 0, meaning there are no controller-specific parameters.
- ❑ **outputdelay** is an optional parameter that specifies the number of output buffers to delay before actually starting the output. The default is 1, meaning that output is double-buffered and does not start until the second SIO\_put call.

## Data Streaming

DAX devices can be opened for input or output. The DAX driver places no restriction on the size or memory segment of the data buffers used when streaming to or from the device (though the associated controller can). Two or more buffers should be used with DAX devices to obtain appropriate buffering for the application.

Tasks block when calling SIO\_get if a full buffer is not available from DAX. Tasks block when calling SIO\_put if an empty buffer is not available from DAX.

## Example

The following example declares DAX\_PRMS as a DAX\_Params structure:

```
#include <dax.h>
#include <cry.h>
DAX_Params DAX_PRMS {
    CRY_bind;
    CRY_ctrl;
    CRY_start;
    CRY_stop;
    CRY_unbind;
    0;
    1;
}
```

By typing \_DAX\_PRMS for the Parameters property of a device, the values above are used as the parameters for this device.

DGN Driver	<i>Software generator driver</i>
<b>Comments</b>	<p>The DGN driver manages a class of software devices known as generators, which produce an input stream of data through successive application of some arithmetic function. DGN devices are used to generate sequences of constants, sine waves, random noise, or other streams of data defined by a user function. The number of active generator devices in the system is limited only by the availability of memory.</p>
<b>Configuring a DGN Device</b>	<p>To add a DGN device, right-click on the DGN - Software Generator Driver icon and select Insert DGN. From the Object menu, choose Rename and type a new name for the DGN device. Open the Properties dialog for the device you created and modify its properties.</p>
<b>Data Streaming</b>	<p>DGN generator devices can be opened for input data streaming only; generators cannot be used as output devices.</p> <p>The DGN driver places no inherent restrictions on the size or memory segment of the data buffers used when streaming from a generator device. Since generators are fabricated entirely in software and do not overlap I/O with computation, no more than one buffer is required to attain maximum performance.</p> <p>Since DGN generates data “on demand,” tasks do not block when calling SIO_get, SIO_put, or SIO_reclaim on a DGN data stream. High-priority tasks must, therefore, be careful when using these streams since lower- or even equal-priority tasks do not get a chance to run until the high-priority task suspends execution for some other reason.</p>
<b>DGN Driver Properties</b>	<p>There are no global properties for the DGN driver manager.</p>
<b>DGN Object Properties</b>	<p>The following properties can be set for a DGN device on the DGN Object Properties dialog in the Configuration Tool:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>comment.</b> Type a comment to identify this object.</li> <li><input type="checkbox"/> <b>Device category.</b> The device category (user, sine, random, constant, printHex, or printInt) determines the type of data stream produced by the device. A sine, random, or constant device can be opened for input data streaming only. A printHex or printInt device can be opened for output data streaming only. <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> <b>user.</b> Uses a custom function to produce or consume a data stream</li> <li><input checked="" type="checkbox"/> <b>sine.</b> Produce a stream of sine wave samples</li> <li><input checked="" type="checkbox"/> <b>random.</b> Produces a stream of random values</li> </ul> </li> </ul>

- **constant.** Produces a constant stream of data
- **printHex.** Writes the stream data buffers to the trace buffer in hexadecimal format
- **printInt.** Writes the stream data buffers to the trace buffer in integer format
- ☐ **Use default parameters.** Check this box if you want to use the default parameters shown in this dialog for the Device category you selected.
- ☐ **Device ID.** This field is set automatically when you select a Device category.
- ☐ **Constant value.** The constant value to be generated if the Device category is constant.
- ☐ **Seed value.** The initial seed value used by an internal pseudo-random number generator if the Device category is random. Used to produce a uniformly distributed sequence of numbers ranging between Lower limit and Upper limit.
- ☐ **Lower limit.** The lowest value to be generated if the Device category is random.
- ☐ **Upper limit.** The highest value to be generated if the Device category is random.
- ☐ **Gain.** The amplitude scaling factor of the generated sine wave if the Device category is sine. The scaling factor is applied to each data point.

To improve performance, the sine wave magnitude (maximum and minimum) value is approximated to the nearest power of two. This is done by computing a shift value by which each entry in the table is right-shifted before being copied into the input buffer. For example, if you set the Gain to 100, the sine wave magnitude is 128, the nearest power of two.

- ❑ **Frequency.** The frequency of the generated sine wave (in cycles per second) if the Device category is sine.

DGN uses a static (256 word) sine table to approximate a sine wave. Only frequencies that divide evenly into 256 can be represented exactly with DGN. A “step” value is computed at open time to be used when stepping through this table:

```
step = (256 * sine.freq / sine.rate)
```

- ❑ **Phase.** The phase of the generated sine wave (in radians) if the Device category is sine.
- ❑ **Sample rate.** The sampling rate of the generated sine wave (in sample points per second) if the Device category is sine.
- ❑ **User function.** If the Device category is user, specifies the function to be used to compute the successive values of the data sequence in an input device, or to be used to process the data stream, in an output device. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- ❑ **User function argument.** An argument to pass to the User function.

A user function must have the following form:

```
fxn(Arg arg, Ptr buf, Uns nmadus)
```

where buf contains the values generated or to be processed. buf and nmadus correspond to the buffer address and buffer size (in MADUs), respectively, for an SIO\_get operation.

**DGS Driver***Stackable gather/scatter driver***Description**

The DGS driver manages a class of stackable devices which compress or expand a data stream by applying a user-supplied function to each input or output buffer. This driver might be used to pack data buffers before writing them to a disk file or to unpack these same buffers when reading from a disk file. All (un)packing must be completed on frame boundaries as this driver (for efficiency) does not maintain remainders across I/O operations.

On opening a DGS device by name, DGS uses the unmatched portion of the string to recursively open an underlying device.

This driver requires a transform function and a packing/unpacking ratio which are used when packing/unpacking buffers to/from the underlying device.

**Configuring a DGS Device**

To add a DGS device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table.** Type `_DGS_FXNS`
- ☐ **Parameters.** Type 0 (zero) to use the default parameters. To use different values, you must declare a `DGS_Params` structure (as described after this list) containing the values to use for the parameters.
- ☐ **Device ID.** Type 0 (zero).
- ☐ **Init Fxn.** Type 0 (zero).
- ☐ **Stacking Device.** Put a checkmark in this box.

`DGS_Params` is defined in `dgs.h` as follows:

```
/* ===== DGS_Params ===== */
typedef struct DGS_Params {          /* device parameters
*/
    Fxn    createFxn;
    Fxn    deleteFxn;
    Fxn    transFxn;
    Arg    arg;
    Int    num;
    Int    den;
} DGS_Params;
```

The device parameters are:

- ❑ **create function.** Optional, default is NULL. Specifies a function that is called to create and/or initialize a transform specific object. If non-NULL, the create function is called in `DGS_open` upon creating the stream with argument as its only parameter. The return value of the create function is passed to the transform function.
- ❑ **delete function.** Optional, default is NULL. Specifies a function to be called when the device is closed. It should be used to free the object created by the create function.
- ❑ **transform function.** Required, default is `localcopy`. Specifies the transform function that is called before calling the underlying device's output function in output mode and after calling the underlying device's input function in input mode. Your transform function should have the following interface:

```
dstsize = myTrans(Arg arg, Void *src, Void *dst, Int srcsize)
```

where `arg` is an optional argument (either argument or created by the create function), and `*src` and `*dst` specify the source and destination buffers, respectively. `srcsize` specifies the size of the source buffer and `dstsize` specifies the size of the resulting transformed buffer (`srcsize * numerator/denominator`).

- ❑ **arg.** Optional argument, default is 0. If the create function is non-NULL, the `arg` parameter is passed to the create function and the create function's return value is passed as a parameter to the transform function; otherwise, argument is passed to the transform function.
- ❑ **num** and **den** (numerator and denominator). Required, default is 1 for both parameters. These parameters specify the size of the transformed buffer. For example, a transformation that compresses two 32-bit words into a single 32-bit word would have `numerator = 1` and `denominator = 2` since the buffer resulting from the transformation is 1/2 the size of the original buffer.

## Transform Functions

The following transform functions are already provided with the DGS driver:

- ❑ **u32tou8/u8tou32.** These functions provide conversion to/from packed unsigned 8-bit integers to unsigned 32-bit integers. The buffer must contain a multiple of 4 number of 32-bit/8-bit unsigned values.
- ❑ **u16tou32/u32tou16.** These functions provide conversion to/from packed unsigned 16-bit integers to unsigned 32-bit integers. The buffer must contain an even number of 16-bit/32-bit unsigned values.



- ❑ **i16toi32/i32toi16.** These functions provide conversion to/from packed signed 16-bit integers to signed 32-bit integers. The buffer must contain an even number of 16-bit/32-bit integers.
- ❑ **u8toi16/i16tou8.** These functions provide conversion to/from a packed 8-bit format (two 8-bit words in one 16-bit word) to a one word per 16 bit format.
- ❑ **i16tof32/f32toi16.** These functions provide conversion to/from packed signed 16-bit integers to 32-bit floating point values. The buffer must contain an even number of 16-bit integers/32-bit Floats.
- ❑ **localcopy.** This function simply passes the data to the underlying device without packing or compressing it.

## Data Streaming

DGS devices can be opened for input or output. DGS\_open allocates buffers for use by the underlying device. For input devices, the size of these buffers is (bufsize \* numerator) / denominator. For output devices, the size of these buffers is (bufsize \* denominator) / numerator. Data is transformed into or out of these buffers before or after calling the underlying device's output or input functions respectively.

You can use the same stacking device in more than one stream, provided that the terminating device underneath it is not the same. For example, if u32tou8 is a DGS device, you can create two streams dynamically as follows:

```
stream = SIO_create("/u32tou8/codec", SIO_INPUT, 128, NULL);  
...  
stream = SIO_create("/u32tou8/port", SIO_INPUT, 128, NULL);
```

You can also create the streams with the Configuration Tool. To do that, add two new SIO objects. Enter /codec (or any other configured terminal device) as the Device Control Parameter for the first stream. Then select the DGS device configured to use u32tou8 in the Device property. For the second stream, enter /port as the Device Control Parameter. Then select the DGS device configured to use u32tou8 in the Device property.

**Example**

The following code example declares DGS\_PRMS as a DGS\_Params structure:

```
#include <dgs.h>

DGS_Params DGS_PRMS {
    NULL,          /* optional create function */
    NULL,          /* optional delete function */
    u32tou8,       /* required transform function */
    0,             /* optional argument */
    4,             /* numerator */
    1              /* denominator */
}
```

By typing `_DGS_PRMS` for the Parameters property of a device, the values above are used as the parameters for this device.

**See Also**

DTR

**DHL Driver***Host link driver***Description**

The DHL driver manages data streaming between the host and the DSP. Each DHL device has an underlying HST object. The DHL device allows the target program to send and receive data from the host through an HST channel using the SIO streaming API rather than using pipes. The DHL driver copies data between the stream's buffers and the frames of the pipe in the underlying HST object.

**Configuring a DHL Device**

To add a DHL device you must first add an HST object and make it available to the DHL driver. Right click on the HST – Host Channel Manager icon and add a new HST object. Open the Properties dialog of the HST object and put a checkmark in the Make this channel available for a new DHL device box. If you plan to use this channel for an output DHL device, make sure that you select output as the mode of the HST channel.

Once there are HST channels available for DHL, right click on the DHL – Host Link Driver icon and select Insert DHL. You can rename the DHL device and then open the Properties dialog to select which HST channel, of those available for DHL, is used by this DHL device. If you plan to use the DHL device for output to the host, be sure to select an HST channel whose mode is output. Otherwise, select an HST channel with input mode.

Note that once you have selected an HST channel to be used by a DHL device, that channel is now owned by the DHL device and is no longer available to other DHL channels.

**Data Streaming**

DHL devices can be opened for input or output data streaming. A DHL device used by a stream created in output mode must be associated with an output HST channel. A DHL device used by a stream created in input mode must be associated with an input HST channel. If these conditions are not met, a SYS\_EBADOBJ error is reported in the system log during startup when the BIOS\_start routine calls the DHL\_open function for the device.

To use a DHL device in a stream created with the Configuration Tool, select the device from the drop-down list in the Device box of its Properties dialog.

To use a DHL device in a stream created dynamically with SIO\_create, use the DHL device name (as it appears in the Configuration Tool) preceded by "/" (forward slash) as the first parameter of SIO\_create:

```
stream = SIO_create("/dh10", SIO_INPUT, 128, NULL);
```

To enable data streaming between the target and the host through streams that use DHL devices, you must bind and start the underlying HST channels of the DHL devices from the Host Channels Control in Code Composer Studio, just as you would with other HST objects.

DHL devices copy the data between the frames in the HST channel's pipe and the stream's buffers. In input mode, it is the size of the frame in the HST channel that drives the data transfer. In other words, when all the data in a frame has been transferred to stream buffers, the DHL device returns the current buffer to the stream's fromdevice queue, making it available to the application. (If the stream buffers can hold more data than the HST channel frames, the stream buffers always come back partially full.) In output mode it is the opposite: the size of the buffers in the stream drives the data transfer so that when all the data in a buffer has been transferred to HST channel frames, the DHL device returns the current frame to the channel's pipe. In this situation, if the HST channel's frames can hold more data than the stream's buffers, the frames always return to the HST pipe partially full.

The maximum performance in a DHL device is obtained when you configure the frame size of its HST channel to match the buffer size of the stream that uses the device. The second best alternative is to configure the stream buffer (or HST frame) size to be larger than, and a multiple of, the size of the HST frame (or stream buffer) size for input (or output) devices. Other configuration settings also work since DHL does not impose restrictions on the size of the HST frames or the stream buffers, but performance is reduced.

## Constraints

- ☐ HST channels used by DHL devices are not available for use with PIP APIs.
- ☐ Multiple streams cannot use the same DHL device. If more than one stream attempts to use the same DHL device, a SYS\_EBUSY error is reported in the system LOG during startup when the BIOS\_start routing calls the DHL\_open function for the device.

## DHL Driver Properties

The following global property can be set for the DHL - Host Link Driver on the DHL Properties dialog in the Configuration Tool:

- ☐ **Object memory.** Enter the memory segment from which to allocate DHL objects. Note that this does not affect the memory segments from where the underlying HST object or its frames are allocated. The memory segment for HST objects and their frames can be set in the HST Manager Properties and HST Object Properties dialogs of the Configuration Tool.

## DHL Object Properties

The following properties can be set for a DHL device using the DHL Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this object.
- ☐ **Underlying HST Channel.** Select the underlying HST channel from the drop-down list. Only HST objects whose properties have a checkmark in the Make this channel available for a new DHL device box are listed
- ☐ **Mode.** This informational property shows the mode (input or output) of the underlying HST channel. This becomes the mode of the DHL device.

**DNL Driver***Null driver***Description**

The DNL driver manages “empty” devices which nondestructively produce or consume data streams. The number of empty devices in the system is limited only by the availability of memory; DNL instantiates a new object representing an empty device on opening, and frees this object when the device is closed.

The DNL driver does not define device ID values or a params structure which can be associated with the name used when opening an empty device. The driver also ignores any unmatched portion of the name declared in the system configuration file when opening a device.

**Configuring a DNL Device**

To add a DNL device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table.** Type `_DNL_FXNS`
- ☐ **Parameters.** Type 0 (zero).
- ☐ **Device ID.** Type 0 (zero).
- ☐ **Init Fxn.** Type 0 (zero).
- ☐ **Stacking Device.** Do not put a checkmark in this box.

**Data Streaming**

DNL devices can be opened for input or output data streaming. Note that these devices return buffers of undefined data when used for input.

The DNL driver places no inherent restrictions on the size or memory segment of the data buffers used when streaming to or from an empty device. Since DNL devices are fabricated entirely in software and do not overlap I/O with computation, no more than one buffer is required to attain maximum performance.

Tasks do not block when using `SIO_get`, `SIO_put`, or `SIO_reclaim` with a DNL data stream.

**DOV Driver***Stackable overlap driver***Description**

The DOV driver manages a class of stackable devices that generate an overlapped stream by retaining the last N minimum addressable data units (MADUs) of each buffer input from an underlying device. These N points become the first N points of the next input buffer. MADUs are equivalent to a 16-bit word in the data address space of the processor on the C54x and C55x platforms and to an 8-bit word on the C6x platforms.

**Configuring a DOV Device**

To add a DOV device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table.** Type `_DOV_FXNS`
- ☐ **Parameters.** Type 0 (zero) or the length of the overlap as described after this list.
- ☐ **Device ID.** Type 0 (zero).
- ☐ **Init Fxn.** Type 0 (zero).
- ☐ **Stacking Device.** Put a checkmark in this box.

If you enter 0 for the Device ID, you need to specify the length of the overlap when you create the stream with `SIO_create` by appending the length of the overlap to the device name. If you create the stream with the Configuration Tool instead, enter the length of the overlap in the Device Control Parameter for the stream.

For example, if you create a device called overlap with the Configuration Tool, and enter 0 as its Device ID, you can open a stream with:

```
stream = SIO_create("/overlap16/codec", SIO_INPUT,
128, NULL);
```

This causes SIO to open a stack of two devices. `/overlap16` designates the device called overlap, and 16 tells the driver to use the last 16 MADUs of the previous frame as the first 16 MADUs of the next frame. `codec` specifies the name of the physical device which corresponds to the actual source for the data.

If, on the other hand you add a device called overlap and enter 16 as its Device ID, you can open the stream with:

```
stream = SIO_create("/overlap/codec", SIO_INPUT, 128, NULL);
```

This causes SIO to open a stack of two devices. /overlap designates the device called overlap, which you have configured to use the last 16 MADUs of the previous frame as the first 16 MADUs of the next frame. As in the previous example, codec specifies the name of the physical device that corresponds to the actual source for the data.

If you create the stream with the Configuration Tool and enter 16 as the Device ID property, leave the Device Control Parameter blank.

In addition to the Configuration Tool properties, you need to specify the value that DOV uses for the first overlap, as in the example:

```
#include <dov.h>

static DOV_Config DOV_CONFIG = {
    (Char) 0
}
DOV_Config *DOV = &DOV_CONFIG;
```

If floating point 0.0 is required, the initial value should be set to (Char) 0.0.

## Data Streaming

DOV devices can only be opened for input.

The overlap size, specified in the string passed to SIO\_create, must be greater than 0 and less than the size of the actual input buffers.

DOV does not support any control calls. All SIO\_ctrl calls are passed to the underlying device.

You can use the same stacking device in more than one stream, provided that the terminating device underneath it is not the same. For example, if overlap is a DOV device with a Device ID of 0:

```
stream = SIO_create("/overlap16/codec", SIO_INPUT, 128, NULL);
...
stream = SIO_create("/overlap4/port", SIO_INPUT, 128, NULL);
```

or if overlap is a DOV device with positive Device ID:

```
stream = SIO_create("/overlap/codec", SIO_INPUT, 128, NULL);
...
stream = SIO_create("/overlap/port", SIO_INPUT, 128, NULL);
```

To create the same streams with the Configuration Tool (rather than dynamically with SIO\_create), add SIO objects with the Configuration Tool. Enter the string that identifies the terminating device preceded by "/" (forward slash) in the SIO object's Device Control Parameters (for example, /codec, /port). Then select the stacking device (overlap, overlapio) from the Device property.

## See Also

DTR  
DGS



**DPI Driver***Pipe driver***Description**

The DPI driver is a software device used to stream data between tasks on a single processor. It provides a mechanism similar to that of UNIX named pipes; a reader and a writer task can open a named pipe device and stream data to/from the device. Thus, a pipe simply provides a mechanism by which two tasks can exchange data buffers.

Any stacking driver can be stacked on top of DPI. DPI can have only one reader and one writer task.

It is possible to delete one end of a pipe with `SIO_delete` and recreate that end with `SIO_create` without deleting the other end.

**Configuring a  
DPI Device**

To add a DPI device, right-click on the DPI - Pipe Driver folder, and select Insert DPI. From the Object menu, choose Rename and type a new name for the DPI device.

**Data Streaming**

After adding a DPI device called `pipe0` in the Configuration Tool, you can use it to establish a communication pipe between two tasks. You can do this dynamically, by calling in the function for one task:

```
inStr = SIO_create("/pipe0", SIO_INPUT, bufsize,
NULL);
...
SIO_get(inStr, bufp);
```

And in the function for the other task:

```
outStr = SIO_create("/pipe0", SIO_OUTPUT, bufsize, NULL);
...
SIO_put(outStr, bufp, nmadus);
```

or by adding with the Configuration Tool two streams that use `pipe0`, one in output mode (`outStream`) and the other one in input mode(`inStream`). Then, from the reader task call:

```
extern SIO_Obj inStream;
SIO_handle inStr = &inStream
...
SIO_get(inStr, bufp);
```

and from the writer task call:

```
extern SIO_Obj outStream;
SIO_handle outStr = &outStream
...
SIO_put(outStr, bufp, nmadus);
```

## DPI and the SIO\_ISSUERECLAIM Streaming Model

The DPI driver places no inherent restrictions on the size or memory segments of the data buffers used when streaming to or from a pipe device, other than the usual requirement that all buffers be the same size.

Tasks block within DPI when using SIO\_get, SIO\_put, or SIO\_reclaim if a buffer is not available. SIO\_select can be used to guarantee that a call to one of these functions do not block. SIO\_select can be called simultaneously by both the input and the output sides.

In the SIO\_ISSUERECLAIM streaming model, an application reclaims the buffers from a stream in the same order as they were previously issued. To preserve this mechanism of exchanging buffers with the stream, the default implementation of the DPI driver for ISSUERECLAIM copies the full buffers issued by the writer to the empty buffers issued by the reader.

A more efficient version of the driver that exchanges the buffers across both sides of the stream, rather than copying them, is also provided. To use this variant of the pipe driver for ISSUERECLAIM, edit the C source file dpi.c provided in the C:\ti\c6000\bios\src\drivers folder. Comment out the following line:

```
#define COPYBUFS
```

Rebuild dpi.c. Link your application with this version of dpi.obj instead of the one in the spoxdev library. To do this, add this version of dpi.obj to your link line explicitly, or add it to a library that is linked ahead of the spoxdev library.

This buffer exchange alters the way in which the streaming mechanism works. When using this version of the DPI driver, the writer reclaims first the buffers issued by the reader rather than its own issued buffers, and vice versa.

This version of the pipe driver is not suitable for applications in which buffers are broadcasted from a writer to several readers. In this situation it is necessary to preserve the ISSUERECLAIM model original mechanism, so that the buffers reclaimed on each side of a stream are the same that were issued on that side of the stream, and so that they are reclaimed in the same order that they were issued. Otherwise, the writer reclaims two or more different buffers from two or more readers, when the number of buffers it issued was only one.

**Converting a Single Processor Application to a Multiprocessor Application**

It is trivial to convert a single-processor application using tasks and pipes into a multiprocessor application using tasks and communication devices. If using SIO\_create, the calls in the source code would change to use the names of the communication devices instead of pipes. (If the communication devices were given names like /pipe0, there would be no source change at all.) If the streams were created with the Configuration Tool instead, you would need to change the Device property for the stream in the configuration template, save and rebuild your application for the new configuration. No source change would be necessary.

**Constraints**

Only one reader and one writer can open the same pipe.

**DPI Driver Properties**

There are no global properties for the DPI driver manager.

**DPI Object Properties**

The following property can be set for a DPI device on the DPI Object Properties dialog in the Configuration Tool:

**comment.** Type a comment to identify this object.

**Allow virtual instances of this device.** Put a checkmark in this box if you want to be able to use SIO\_create to dynamically create multiple streams that will use this DPI device. DPI devices are used by SIO stream objects, which you create with the DSP/BIOS Configuration Tool or the SIO\_create function.

If this box is checked, when you use SIO\_create, you can create multiple streams that use the same DPI driver by appending numbers to the end of the name. For example, if the DPI object is named "pipe", you can call SIO\_create to create pipe0, pipe1, and pipe2. Only integer numbers can be appended to the name.

If this box is not checked, when you use SIO\_create, the name of the SIO object must exactly match the name of the DPI object. As a result, only one open stream can use the DPI object. For example, if the DPI object is named "pipe", an attempt to use SIO\_create to create pipe0 will fail.

**DST Driver***Stackable split driver***Description**

This stacking driver can be used to input or output buffers that are larger than the physical device can actually handle. For output, a single (large) buffer is split into multiple smaller buffers which are then sent to the underlying device. For input, multiple (small) input buffers are read from the device and copied into a single (large) buffer.

**Configuring a DST Device**

To add a DST device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table.** Type `_DST_FXNS`
- ☐ **Parameters.** Type 0 (zero).
- ☐ **Device ID.** Type 0 (zero) or the number of small buffers corresponding to a large buffer as described after this list.
- ☐ **Init Fxn.** Type 0 (zero).
- ☐ **Stacking Device.** Put a checkmark in this box.

If you enter 0 for the Device ID, you need to specify the number of small buffers corresponding to a large buffer when you create the stream with `SIO_create`, by appending it to the device name.

**Example 1:**

For example, if you create a user-defined device called `split` with the Configuration Tool, and enter 0 as its Device ID property, you can open a stream with:

```
stream = SIO_create("/split4/codec", SIO_INPUT, 1024, NULL);
```

This causes SIO to open a stack of two devices: `/split4` designates the device called `split`, and 4 tells the driver to read four 256-word buffers from the `codec` device and copy the data into 1024-word buffers for your application. `codec` specifies the name of the physical device which corresponds to the actual source for the data.

Alternatively, you can create the stream with the Configuration Tool (rather than by calling `SIO_create` at run-time). To do so, first create and configure two user-defined devices called `split` and `codec`. Then, create an SIO object. Type `4/codec` as the Device Control Parameter. Select `split` from the Device list.

**Example 2:**

Conversely, you can open an output stream that accepts 1024-word buffers, but breaks them into 256-word buffers before passing them to /codec, as follows:

```
stream = SIO_create("/split4/codec", SIO_OUTPUT, 1024, NULL);
```

To create this output stream with the Configuration Tool, you would follow the steps for example 1, but would select output for the Mode property of the SIO object.

**Example 3:**

If, on the other hand, you add a device called split and enter 4 as its Device ID, you need to open the stream with:

```
stream = SIO_create("/split/codec", SIO_INPUT, 1024, NULL);
```

This causes SIO to open a stack of two devices: /split designates the device called split, which you have configured to read four buffers from the codec device and copy the data into a larger buffer for your application. As in the previous example, codec specifies the name of the physical device that corresponds to the actual source for the data.

When you type 4 as the Device ID, you do not need to type 4 in the Device Control Parameter for an SIO object created with the Configuration Tool. Type only/codec for the Device Control Parameter.

**Data Streaming**

DST stacking devices can be opened for input or output data streaming.

**Constraints**

- ☐ The size of the application buffers must be an integer multiple of the size of the underlying buffers.
- ☐ This driver does not support any SIO\_ctrl calls.

**DTR Driver***Stackable streaming transformer driver***Description**

The DTR driver manages a class of stackable devices known as transformers, which modify a data stream by applying a function to each point produced or consumed by an underlying device. The number of active transformer devices in the system is limited only by the availability of memory; DTR instantiates a new transformer on opening a device, and frees this object when the device is closed.

Buffers are read from the device and copied into a single (large) buffer.

**Configuring a DTR Device**

To add a DTR device, right-click on the User-defined Devices icon in the Configuration Tool, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- ☐ **DEV\_FXNS table.** Type `_DTR_FXNS`
- ☐ **Parameters.** Enter the name of a `DTR_Params` structure declared in your C application code. See the information following this list for details.
- ☐ **Device ID.** Type 0 (zero) or `_DTR_multiply`.

If you type 0, you need to supply a user function in the device parameters. This function is called by the driver as follows to perform the transformation on the data stream:

```
if (user.fxn != NULL) {
    (*user.fxn)(user.arg, buffer, size);
}
```

If you type `_DTR_multiply`, a data scaling operation is performed on the data stream to multiply the contents of the buffer by the `scale.value` of the device parameters.

- ☐ **Init Fxn.** Type 0 (zero).
- ☐ **Stacking Device.** Put a checkmark in this box.

The DTR\_Params structure is defined in dtr.h as follows:

```
/* ===== DTR_Params ===== */
typedef struct { /* device parameters */
    struct {
        DTR_Scale value; /* scaling factor */
    } scale;
    struct {
        Arg arg; /* user-defined argument */
        Fxn fxn; /* user-defined function */
    } user;
} DTR_Params;
```

In the following code example, DTR\_PRMS is declared as a DTR\_Params structure:

```
#include <dtr.h>
...
struct DTR_Params DTR_PRMS = {
    10.0,
    NULL,
    NULL
};
```

By typing \_DTR\_PRMS as the Parameters property of a DTR device, the values above are used as the parameters for this device.

You can also use the default values that the driver assigns to these parameters by entering \_DTR\_PARAMS for this property. The default values are:

```
DTR_Params DTR_PARAMS = {
    { 1 }, /* scale.value */
    { (Arg)NULL, /* user.arg */
      (Fxn)NULL }, /* user.fxn */
};
```

scale.value is a floating-point quantity multiplied with each data point in the input or output stream.

user.fxn and user.arg define a transformation that is applied to inbound or outbound blocks of data, where buffer is the address of a data block containing size points; if the value of user.fxn is NULL, no transformation is performed at all.

```
if (user.fxn != NULL) {
    (*user.fxn)(user.arg, buffer, size);
}
```

**Data Streaming**

DTR transformer devices can be opened for input or output and use the same mode of I/O with the underlying streaming device. If a transformer is used as a data source, it inputs a buffer from the underlying streaming device and then transforms this data in place. If the transformer is used as a data sink, it outputs a given buffer to the underlying device after transforming this data in place.

The DTR driver places no inherent restrictions on the size or memory segment of the data buffers used when streaming to or from a transformer device; such restrictions, if any, would be imposed by the underlying streaming device.

Tasks do not block within DTR when using SIO. A task can, of course, block as required by the underlying device.



## 2.6 Global Settings

This module is the global settings manager.

### Functions

None

### Description

This module does not manage any individual objects, but rather allows you to control global or system-wide settings used by other modules.

### Global Settings Properties

The following Global Settings can be made:

- ☐ **Target Board Name.** The type of board on which your target device is mounted.
- ☐ **DSP Speed In MHz (CLKOUT).** This number, times 1000000, is the number of instructions the processor can execute in 1 second. This value is used by the CLK manager to calculate register settings for the on-device timers.
- ☐ **DSP Type.** Target CPU family. Specifies which family of C5x is being used. It is normally unwritable, and is controlled by the Chip Support Library (CSL) property. When the CSL is specified as other, this field becomes writable.
- ☐ **Chip Support Library (CSL).** Specifies the specific chip type, such as 5402, 5440, 5510, etc. This controls which CSL library is linked with the application and also controls the DSP Type property. Select other to remove support for the CSL and to allow you to select a DSP family in the DSP Type field.
- ☐ **Chip Support Library Name.** Specifies the name of the CSL library that will be linked with the application. This property is informational only. It is not writable.
- ☐ **PMST(6-0).** The low seven bits of the PMST register (MP/MC, OVLY, AVIS, DROM, CLKOFF, SMUL, and SST). Only the low seven bits can be directly modified. The high nine bits (IPTR) of the PMST are computed based on the base address of the VECT memory segment.
- ☐ **PMST(15-0).** The entire PMST register. PMST(6-0) can be modified directly. PMST(15-7) are computed based on the base address of the VECT memory segment.
- ☐ **SWWSR.** The value for the Software Wait-State Register, which controls the software-programmable wait-state generator.



The SWWSR, BSCR, and CLKMD registers are initialized during the boot initialization (via BIOS\_init) before the program's main function is called. See *Volume 1: CPU and Peripherals* of the TMS320C6000 DSP Reference Set for details on the SWWSR, BSCR, and CLKMD.



- ☐ **BSCR.** The value for the Bank-Switching Control Register, which allows switching between external memory banks without requiring external wait states.
- ☐ **Modify CLKMD.** Put a checkmark in this box if you want to modify the value of the Clock Mode Register, which is used to program the PLL (phase-locked loop).



- ☐ **CLKMD - (PLL) Clock Mode Register.** The value of the Clock Mode Register.



- ☐ **Function Call Model.** This setting controls which libraries are used to link the application. If you change this setting, you must set the compiler and linker options to correspond. Use the far option only with C54x devices that support extended addressing (for example, 5402, 549, 5410).
- ☐ **Memory Model.** This specifies the address reach within the program. The options are small and large. In the small model, the data address is limited to 16-bit addressing. In the large model, the data addressability is the full 23-bit range. Program space addressing is always the full 24-bit range.
- ☐ **Call User Init Function.** Put a checkmark in this box if you want an initialization function to be called early during program initialization, after .cinit processing and before the main function. This function can perform special hardware setup. The code in this function should not use any DSP/BIOS API calls.
- ☐ **User Init Function.** Type the name of the initialization function.
- ☐ **Enable Real Time Analysis.** Remove the checkmark from this box if you want to remove support for DSP/BIOS implicit instrumentation from the program. This optimizes a program by reducing code size, but removes support for the analysis tools and the LOG, STS, and TRC module APIs.
- ☐ **Enable All TRC Trace Event Classes.** Remove the checkmark from this box if you want all types of tracing to be initially disabled when the program is loaded. If you disable tracing, you can still use the RTA Control Panel or the TRC\_enable function to enable tracing at run-time.

## 2.7 HST Module

The HST module is the host channel manager.

### Functions

- ☐ **HST\_getpipe.** Get corresponding pipe object

### Description

The HST module manages host channel objects, which allow an application to stream data between the target and the host. Host channels are statically configured for input or output. Input channels (also called the source) read data from the host to the target. Output channels (also called the sink) transfer data from the target to the host.

---

**Note:**

HST channel names cannot begin with a leading underscore ( \_ ).

---

Each host channel is internally implemented using a data pipe (PIP) 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).

During early development, especially when testing software interrupt processing algorithms, programs can use host channels to input canned data sets and to output the results. Once the algorithm appears sound, you can replace these host channel objects with I/O drivers for production hardware built around DSP/BIOS pipe objects. By attaching host channels as probes to these pipes, you can selectively capture the I/O channels in real time for off-line and field-testing analysis.

The notify function is called in the context of the code that calls PIP\_free or PIP\_put. This function can be written in C or assembly. The code that calls PIP\_free or PIP\_put should preserve any necessary registers.

The other end of the host channel is managed by the LNK\_dataPump IDL object. Thus, a channel can only be used when some CPU capacity is available for IDL thread execution.

### HST Manager Properties

The following global properties can be set for the HST module on the HST Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment containing HST objects.
- ☐ **Host Link Type.** The underlying physical link to be used for host-target data transfer. If None is selected, no instrumentation or host channel data is transferred between the target and host in real time. The Analysis Tool windows are updated only when the target is

halted (for example, at a breakpoint). The program code size is smaller when the Host Link Type is set to None because RTDX code is not included in the program.

## HST Object Properties

A host channel maintains a buffer partitioned into a fixed number of fixed length frames. All I/O operations on these channels deal with one frame at a time; although each frame has a fixed length, the application can put a variable amount of data in each frame.

The following properties can be set for a host file object on the HST Object Properties dialog in the Configuration Tool:

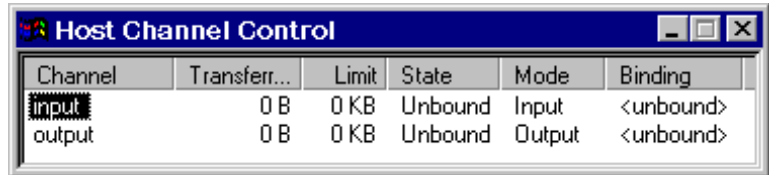
- ☐ **comment.** Type a comment to identify this HST object.
- ☐ **mode.** The type of channel: input or output. Input channels are used by the target to read data from the host; output channels are used by the target to transfer data from the target to the host.
- ☐ **bufseg.** The memory segment from which the buffer is allocated; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- ☐ **bufalign.** The alignment (in words) of the buffer allocated within the specified memory segment.
- ☐ **framesize.** The length of each frame (in words)
- ☐ **numframes.** The number of frames
- ☐ **statistics.** Check this box if you want to monitor this channel with an STS object. You can display the STS object for this channel to see a count of the number of frames transferred with the Statistics View Analysis Tool.
- ☐ **Make this channel available for a new DHL device.** Check this box if you want to use this HST object with a DHL device. DHL devices allow you to manage data I/O between the host and target using the SIO module, rather than the PIP module. See the DHL driver topic for more details.
- ☐ **notify.** The function to execute when a frame of data for an input channel (or free space for an output channel) is available. To avoid problems with recursion, this function should not directly call any of the PIP module functions for this HST object.
- ☐ **arg0, arg1.** Two 16-bit arguments passed to the notify function. They can be either unsigned 16-bit constants or symbolic labels.
- ☐ **arg0, arg1.** Two Arg type arguments passed to the notify function.



## HST - Host Channel Control Interface

If you are using host channels, use the Host Channel Control to bind each channel to a file on your host computer and start the channels.

- 1) Choose the DSP/BIOS→Host Channel Control menu item. You see a window that lists your host input and output channels.



- 2) Right-click on a channel and choose Bind from the pop-up menu.
- 3) Select the file to which you want to bind this channel. For an input channel, select the file that contains the input data. For an output channel, you can type the name of a file that does not exist or choose any file that you want to overwrite.
- 4) Right-click on a channel and choose Start from the pop-up menu. For an input channel, this causes the host to transfer the first frame of data and causes the target to run the function for this HST object. For an output channel, this causes the target to run the function for this HST object.

**HST\_getpipe***Get corresponding pipe object***C Interface**

<b>Syntax</b>	pipe = HST_getpipe(hst);	
<b>Parameters</b>	HST_Handle hst	/* host object handle */
<b>Return Value</b>	PIP_Handle pip	/* pipe object handle*/

**Assembly Interface**

<b>Syntax</b>	HST_getpipe
<b>Preconditions</b>	ar2 = address of the host channel object
<b>Postconditions</b>	ar2 = address of the pipe object
<b>Modifies</b>	ar2, c

**Assembly Interface**

<b>Syntax</b>	HST_getpipe
<b>Preconditions</b>	xar0 = address of the host channel object
<b>Postconditions</b>	xar0 = address of the pipe object
<b>Modifies</b>	xar0

<b>Reentrant</b>	yes
------------------	-----

<b>Description</b>	HST_getpipe gets the address of the pipe object for the specified host channel object.
--------------------	--

**Example**

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

    in = HST_getpipe(input);
    out = HST_getpipe(output);
```

```
    if (PIP_getReaderNumFrames == 0 ||
        PIP_getWriterNumFrames == 0) {
        error;
    }

    /* get input data and allocate output frame */
    PIP_get(in);
    PIP_alloc(out);

    /* copy input data to output frame */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);

    size = PIP_getReaderSize();
    out->writerSize = size;

    for (; size > 0; size--) {
        *dst++ = *src++;
    }

    /* output copied data and free input frame */
    PIP_put(out);
    PIP_free(in);
}
```

**See Also**

PIP\_alloc  
PIP\_free  
PIP\_get  
PIP\_put

## 2.8 HWI Module

The HWI module is the hardware interrupt manager.

### Functions

- ❑ `HWI_disable`. Disable hardware interrupts
- ❑ `HWI_dispatchPlug`. Plug the HWI dispatcher
- ❑ `HWI_enable`. Enable hardware interrupts
- ❑ `HWI_enter`. Hardware ISR prolog
- ❑ `HWI_exit`. Hardware ISR epilog
- ❑ `HWI_restore`. Restore hardware interrupt state

### Description

The HWI module manages hardware interrupts. Using the Configuration Tool, you can assign routines that run when specific hardware interrupts occur. Some routines are assigned to interrupts automatically by the HWI module. For example, the interrupt for the timer that you select for the CLK global properties is automatically configured to run a function that increments the low-resolution time. See the CLK module for more details.

You can also dynamically assign routines to interrupts at run-time using the `HWI_dispatchPlug` function or the `C54_plug` or `C55_plug` function.

Interrupt routines can be written completely in assembly, completely in C, or in a mix of assembly and C. In order to support interrupt routines written completely in C, an HWI dispatcher is provided that performs the requisite prolog and epilog for an interrupt routine.

The HWI dispatcher is the preferred method for handling an interrupt.

When an HWI object does not use the dispatcher, the `HWI_enter` assembly macro must be called prior to any DSP/BIOS API calls that affect other DSP/BIOS objects, such as posting an SWI or a semaphore, and the `HWI_exit` assembly macro must be called at the very end of the function's code.

When an HWI object is configured to use the dispatcher, the dispatcher handles the `HWI_enter` prolog and the `HWI_exit` epilog, and the HWI function can be completely written 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 allows you to save code space by including only one instance of the `HWI_enter`/`HWI_exit` code.



---

**Note:**

CLK functions should not call HWI\_enter and HWI\_exit as these are called internally by DSP/BIOS when it runs CLK\_F\_isr. Additionally, CLK functions should **not** use the *interrupt* keyword or the INTERRUPT pragma in C functions.

---



Whether a hardware interrupt is dispatched by the HWI dispatcher or handled with the HWI\_enter/HWI\_exit macros, a common interrupt stack (called the system stack) is used for the duration of the HWI. This same stack is also used by all SWI routines.



If the task manager is enabled in the DSP/BIOS application, the hardware and software interrupts are handled on the current task stack, otherwise they are handled on the system stack.

In the following notes, references to the usage of HWI\_enter/HWI\_exit also apply to usage of the HWI dispatcher since, in effect, the dispatcher calls HWI\_enter/HWI\_exit.

---

**Note:**

Do not call SWI\_disable or SWI\_enable within an HWI function.

---

---

**Note:**

Do not call HWI\_enter, HWI\_exit, or any other DSP/BIOS functions from a non-maskable interrupt (NMI) service routine.

---

---

**Note:**

Do not call HWI\_enter/HWI\_exit from a HWI function that is invoked by the dispatcher.

---

The DSP/BIOS API calls that require an HWI function to use HWI\_enter and HWI\_exit are:

- ☐ SWI\_andn
- ☐ SWI\_andnHook
- ☐ SWI\_dec
- ☐ SWI\_inc
- ☐ SWI\_or
- ☐ SWI\_orHook
- ☐ SWI\_post

- ☐ PIP\_alloc
- ☐ PIP\_free
- ☐ PIP\_get
- ☐ PIP\_put
- ☐ PRD\_tick
- ☐ SEM\_post
- ☐ MBX\_post
- ☐ TSK\_yield
- ☐ TSK\_tick

---

**Note:**

Any PIP API call can cause the pipe's notifyReader or notifyWriter function to run. If an HWI function calls a PIP function, the notification functions run as part of the HWI function.

---

---

**Note:**

An HWI function must use HWI\_enter and HWI\_exit or must be dispatched by the HWI dispatcher if it indirectly runs a function containing any of the API calls listed above.

---

If your HWI function and the functions it calls do not call any of these API operations, you do not need to disable software interrupt scheduling by calling HWI\_enter and HWI\_exit.

The register mask argument to HWI\_enter and HWI\_exit allows you to save and restore registers used within the function. Other arguments allow the HWI to control the settings of the IMR or, in the case of the C55x device, the IER0[IER1].

Hardware interrupts always interrupt software interrupts unless hardware interrupts have been disabled with HWI\_disable.

---

**Note:**

By using HWI\_enter and HWI\_exit as an HWI function's prolog and epilog, an HWI function can be interrupted; that is, a hardware interrupt can interrupt another interrupt. For the c54x device, you can use the IMRDISABLEMASK parameter for the HWI\_enter API to prevent this from occurring. For the c55x device, you can use the IER0DISABLEMASK and IER1DISABLEMASK parameters to prevent this from occurring.

---

## HWI Manager Properties



DSP/BIOS manages the hardware interrupt vector table and provides basic hardware interrupt control functions; for example, enabling and disabling the execution of hardware interrupts.

The following global property can be set for the HWI module on the HWI Manager Properties dialog in the Configuration Tool:

- ☐ **Stack Mode.** Select the Stack Mode used for the application: C54X\_STK, USE\_RETA or NO\_RETA.

There are no global properties for the HWI manager for the C54x platform.

## HWI Object Properties

The following properties can be set for a hardware interrupt service routine object on the HWI Object Properties dialog in the Configuration Tool:

- ☐ **comment** A comment is provided to identify each HWI object.
- ☐ **function** The function to execute. Interrupt routines that use the dispatcher can be written completely in C or any combination of assembly and C but must not call the HWI\_enter/HWI\_exit macro pair. Interrupt routines that don't use the dispatcher must be written at least partially in assembly language. Within an HWI function that does not use the dispatcher, the HWI\_enter assembly macro must be called prior to any DSP/BIOS API calls that affect other DSP/BIOS objects, such as posting an SWI or a semaphore. HWI functions can post software interrupts, but they do not run until your HWI function (or the dispatcher) calls the HWI\_exit assembly macro, which must be the last statement in any HWI function that calls HWI\_enter.
- ☐ **monitor** If set to anything other than Nothing, an STS object is created for this HWI that is passed the specified value on every invocation of the interrupt service routine. The STS update occurs just before entering the HWI routine.

Be aware that when the monitor property is enabled for a particular HWI object, a code preamble is inserted into the HWI routine to make this monitoring possible. The overhead for monitoring is 20 to 30 instructions per interrupt, per HWI object monitored. Leaving this instrumentation turned on after debugging is not recommended, since HWI processing is the most time-critical part of the system.

- ☐ **addr** If the monitor field above is set to Data Address, this field lets you specify a data memory address to be read; the word-sized value is read and passed to the STS object associated with this HWI object.

- ❑ **type.** The type of the value to be monitored: unsigned or signed. Signed quantities are sign extended when loaded into the accumulator; unsigned quantities are treated as word-sized positive values.
- ❑ **operation.** The operation to be performed on the value monitored. You can choose one of several STS operations.
- ❑ **Use Dispatcher.** A check box that controls whether the HWI dispatcher is used.
- ❑ **Arg.** This argument is passed to the function as its only parameter. You can use either a literal integer or a symbol defined by the application. This property is available only when using the HWI dispatcher.
- ❑ **Interrupt Mask.** A drop-down menu that specifies which interrupts the dispatcher should disable before calling the function. This property is available only when using the HWI dispatcher. (For the C55x platform, separate fields are provided for IER0 and IER1)
- ❑ **Interrupt Bit Mask.** An integer field that is writable when the interrupt mask is set as bitmask. This should be a hexadecimal integer bitmask specifying the interrupts to disable. (For the C55x platform, separate fields are provided for IER0 and IER1)

Although it is not possible to create new HWI objects, most interrupts supported by the device architecture have a precreated HWI object. Your application can require that you select interrupt sources other than the default values in order to rearrange interrupt priorities or to select previously unused interrupt sources.

In addition to the precreated HWI objects, some HWI objects are preconfigured for use by certain DSP/BIOS modules. For example, the CLK module configures an HWI object.

Table 2-1 and Table 2-2 list these precreated objects and their default interrupt sources. The HWI object names are the same as the interrupt names.

Table 2-1. HWI interrupts for the C54x



Name	intrid	Interrupt Type
HWI_RS	0	Reset interrupt.
HWI_NMI	1	Nonmaskable interrupt.
HWI_SINT17-30	2-15	User-defined software interrupts #17 through #30. These interrupt service routines are only triggered by the intr instruction from within the application. These software interrupts are executed immediately upon being triggered.
HWI_INT0	16	External user interrupt #0.
HWI_INT1	17	External user interrupt #1.
HWI_INT2	18	External user interrupt #2.
HWI_TINT	19	Internal timer interrupt.
HWI_SINT4	20	Serial port A receive interrupt.
HWI_SINT5	21	Serial port A transmit interrupt.
HWI_SINT6	22	Serial port B receive interrupt.
HWI_SINT7	23	Serial port B transmit interrupt.
HWI_INT3	24	External user interrupt #3.
HWI_HPIINT	25	Host port interface interrupt.
HWI_BRINT1	26	Buffered serial port receive interrupt
HWI_BXINT1	27	Buffered serial port transmit interrupt

Table 2-2. HWI interrupts for the 'C55x



Name	Interrupt Type
HWI_RESET	Reset interrupt.
HWI_NMI	Nonmaskable interrupt.
HWI_INT2	Maskable (IER0, bit2) hardware interrupt.
HWI_INT3	Maskable (IER0, bit3) hardware interrupt.
HWI_TINT	Timer interrupt. (IER, bit4)
HWI_INT5 through HWI_INT15	Maskable (IER0, bit5) hardware interrupt through Maskable (IER0, bit15) hardware interrupt.
HWI_INT16 through HWI_INT23	Maskable (IER1, bit0) hardware interrupt through Maskable (IER1, bit7) hardware interrupt.
HWI_INT24	Maskable (IER1, bit8) bus error interrupt.
HWI_INT25	Maskable (IER1, bit9) data log interrupt.
HWI_INT26	Maskable (IER1, bit10) RTOS interrupt.
HWI_SINT27	Non maskable software interrupt.
HWI_SINT28	Non maskable software interrupt.
HWI_SINT29	Non maskable software interrupt.
HWI_SINT30	Non maskable software interrupt.
HWI_SINT31	Non maskable software interrupt.

### HWI - Execution Graph Interface

Time spent performing HWI functions is not directly traced for performance reasons. However, if you configure the HWI object properties to perform any STS operations on a register, address, or pointer, you can track time spent performing HWI functions in the Statistics View window, which you can open by choosing DSP/BIOS→Statistics View.

HWI\_disable

Disable hardware interrupts

C Interface

Syntax	oldST1 = HWI_disable();
Parameters	Void
Return Value	Uns oldST1;

Assembly Interface



Syntax	HWI_disable or HWI_disable var
Preconditions	none
Postconditions	intm = 1 a = old value of ST1 (if var specified)
Modifies	intm, a

Assembly Interface



Syntax	HWI_disable
Preconditions	none
Postconditions	intm = 1
Modifies	intm

Reentrant	yes
-----------	-----

Description	<p>HWI_disable disables hardware interrupts by setting the intm bit in the status register. Call HWI_disable before a portion of a function that needs to run without interruption. When critical processing is complete, call HWI_restore or HWI_enable to reenale hardware interrupts.</p> <p>Interrupts that occur while interrupts are disabled are postponed until interrupts are reenaled. However, if the same type of interrupt occurs several times while interrupts are disabled, the interrupt's function is executed only once when interrupts are reenaled.</p>
-------------	--

A context switch can occur when calling HWI\_enable or HWI\_restore if an enabled interrupt occurred while interrupts are disabled.

The Flag parameter is optional. It may be any character(s), and if specified, oldST1 will be returned in register A. If Flag is not specified, there is no return value.

**Constraints and  
Calling Context**

- ❑ HWI\_disable cannot be called from the program's main function.

**Example**

```
old = HWI_disable();  
    'do some critical operation'  
HWI_restore(old);
```

**See Also**

HWI\_enable  
HWI\_restore  
SWI\_disable  
SWI\_enable



HWI\_dispatchPlug

Plug the HWI dispatcher

C Interface

Syntax	HWI_dispatchPlug(vecid, fxn, attrs);
Parameters	<div>Int          vecid;     /* interrupt id */ Fxn          fxn;      /* pointer to HWI function */ HWI_Attrs    *attrs     /*pointer to HWI dispatcher attributes */</div>
Return Value	void
Assembly Interface	none
Reentrant	yes
Description	<p>HWI_dispatchPlug writes four instruction words into the Interrupt-Vector Table, at the address corresponding to vecid. The instructions written in the Interrupt-Vector Table create a call to the HWI dispatcher.</p> <p>The HWI dispatcher table gets filled with the function specified by the fxn parameter and the attributes specified by the attrs parameter.</p> <p>HWI_dispatchPlug does not enable the interrupt. Use C54_enableIMR or C55_enableIER0/C55_enableIER1 to enable specific interrupts.</p> <p>If attrs is NULL, the HWI's dispatcher properties are assigned a default set of attributes. Otherwise, the HWI's dispatcher properties are specified by a structure of type HWI_Attrs defined as follows:</p> <pre>typedef struct HWI_Attrs {     Uns  intrMask;     /* IMR bitmask, 1="self" (default)*/     Arg  arg;          /* fxn arg (default = 0)*/ } HWI_Attrs;</pre> <p>The intrMask element is a bitmask that specifies which interrupts to mask off while executing the HWI. Bit positions correspond to those of the IMR. A value of 1 indicates an interrupt is being plugged. The default value is 1.</p> <p>The arg element is a generic argument that is passed to the plugged function as its only parameter. The default value is 0.</p>
Constraints and Calling Context	<div><div></div><div>vecid must be a valid interrupt ID in the range of 0-31.</div></div>
See Also	<div>HWI_enable HWI_restore SWI_disable SWI_enable</div>

**HWI\_enable***Enable interrupts***C Interface**

<b>Syntax</b>	HWI_enable();
<b>Parameters</b>	Void
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	HWI_enable
<b>Preconditions</b>	none
<b>Postconditions</b>	intm = 0
<b>Modifies</b>	intm

**Assembly Interface**

<b>Syntax</b>	HWI_enable
<b>Preconditions</b>	none
<b>Postconditions</b>	intm = 0
<b>Modifies</b>	intm, tc1

**Reentrant**                      yes

**Description**                      HWI\_enable enables hardware interrupts by clearing the intm bit in the status register.

Hardware interrupts are enabled unless a call to HWI\_disable disables them.

Interrupts that occur while interrupts are disabled are postponed until interrupts are reenabled. However, if the same type of interrupt occurs several times while interrupts are disabled, the interrupt's function is executed only once when interrupts are reenabled. A context switch can

occur when calling HWI\_enable/HWI\_restore if an enabled interrupt occurs while interrupts are disabled.

Any call to HWI\_enable enables interrupts, even if HWI\_disable has been called several times.

### Constraints and Calling Context

- ❑ HWI\_enable cannot be called from the program's main function.

### Example

```
HWI_disable();  
"critical processing takes place"  
HWI_enable();  
"non-critical processing"
```

### See Also

HWI\_disable  
HWI\_restore  
SWI\_disable  
SWI\_enable

**HWI\_enter***Hardware ISR prolog***C Interface**

<b>Syntax</b>	none
<b>Parameters</b>	none
<b>Return Value</b>	none

**Assembly Interface**

<b>Syntax</b>	HWI_enter MASK, IMRDISABLEMASK
<b>Preconditions</b>	intm = 1
<b>Postconditions</b>	dp = GBL_A_SYSPAGE cpl = ovm = c16 = frct = cmpt = braf = arp = 0 intm = 0 sp = even address
<b>Modifies</b>	c, cpl, dp, sp

**Assembly Interface**

<b>Syntax</b>	HWI_enter AR_T_MASK, AC_MASK, MISC1_MASK, MISC2_MASK, MISC3_MASK, IER0DISABLEMASK, IER1DISABLEMASK
<b>Preconditions</b>	intm = 1
<b>Postconditions</b>	braf=0, cpl=1, m40=0, satd=0, sxmd=0, c16=0, frct=0, c54cm=0, arms=1, rdm=0, cdplc=0, ar[0...7]lc=0, sata=0, smul=0, sst=0 Both the user stack pointer (XSP) and the system stack pointer (XSSP) are left aligned to even address boundaries in compliance with standard C conventions.
<b>Modifies</b>	xar0, xar1, ac0g, ac0h, ier0, ier1
<b>Reentrant</b>	yes

<b>Description</b>	HWI_enter is an API (assembly macro) used to save the appropriate context for a DSP/BIOS interrupt service routine (ISR).
--------------------	---

HWI\_enter is used by ISRs that are user-dispatched, as opposed to ISRs that are handled by the HWI dispatcher. HWI\_enter must not be issued by ISRs that are handled by the HWI dispatcher.

If the HWI dispatcher is not used by an HWI object, HWI\_enter must be used in the ISR before any DSP/BIOS API calls that could trigger other DSP/BIOS objects, such as posting an SWI or semaphore. HWI\_enter is used in tandem with HWI\_exit to ensure that the DSP/BIOS SWI or TSK manager is called at the appropriate time. Normally, HWI\_enter and HWI\_exit must surround all statements in any DSP/BIOS assembly language ISRs that call C functions.



One common mask, C54\_CNOTPRESERVED, is defined in c54.h54. This mask specifies the C temporary registers and should be used when saving the context for an ISR that is written in C.

The following are the definitions of the masks specified above:

- ☐ **AR\_T\_MASK.** Mask of registers belonging to ar0-7, t0-3, sp-ssp
- ☐ **AC\_MASK.** Mask of registers belonging to ac0-3
- ☐ **MISC1\_MASK.** Mask of registers ier0, ifr0, dbier0, ier1, ifr, dbier1, st0, st1, st2, st3, trn0, bk03, brc0
- ☐ **MISC2\_MASK.** Mask of registers dp, cdp, mdp, mdp05, mdp67, pdp, bk47, bkc, bof01, bof23, bof45, bof67, bofc, ivpd, ivph, trn1
- ☐ **MISC3\_MASK.** Mask of registers brc1, csr, rsa0\_h\_addr, rsa0, rea0\_h\_addr, rea0, rsa1\_h\_addr, rsa1, rea1\_h\_addr, rea1, rptc
- ☐ **IER0DISABLEMASK.** Mask of ier0 bits to turn off
- ☐ **IER1DISABLEMASK.** Mask of ier1 bits to turn off

---

**Note:**

The macros C55\_saveCcontext, C55\_restoreCcontext, C55\_saveBiosContext, and C55\_restoreBiosContext preserve processor register context per C and DSP/BIOS requirements, respectively.

---

## Constraints and Calling Context

- ☐ This API should not be used for the NMI HWI function.
- ☐ This API must not be called if the HWI object that runs this function uses the HWI dispatcher.
- ☐ This API cannot be called from the program's main function.
- ☐ This API cannot be called from a SWI, TSK, or IDL function.

- ❑ This API cannot be called from a CLK function.
- ❑ Unless the HWI dispatcher is used, this API must be called within any hardware interrupt function (except NMI's HWI function) before the first operation in an ISR that uses any DSP/BIOS API calls that might post or affect a software interrupt or semaphore. Such functions must be written in assembly language. Alternatively, the HWI dispatcher can be used instead of this API, allowing the function to be written completely in C and allowing you to reduce code size.
- ❑ If an interrupt function calls HWI\_enter, it must end by calling HWI\_exit.
- ❑ Do not use the interrupt keyword or the INTERRUPT pragma in C functions that run in the context of an HWI.
- ❑ On the C54x platform, the postconditions of HWI\_enter do not completely satisfy C function calling conventions. Specifically, after calling HWI\_enter, the CPL bit is not set. Before calling a C function, you must set the CPL bit and restore it afterwards to satisfy a precondition for calling HWI\_exit. If using the HWI dispatcher, this is not necessary as the dispatcher assumes a C function and performs this step prior to calling the user function.



### Example



CLK\_isr:

```
HWI_enter C54_CNOTPRESERVED, 0008h
PRD_tick
HWI_exit C54_CNOTPRESERVED, 0008h
```

### Examples



Example #1:

```
.include c55.h55
```

```
AR_T_MASK_clk      .set C55_AR57_MASK
AC_MASK_clk        .set C55_AC3_MASK
MISC1_MASK_clk     .set 0
MISC2_MASK_clk     .set C55_TRN1
MISC3_MASK_clk     .set 0
IER0DISABLEMASK_clk .set 0008h
IER1DISABLEMASK_clk .set 0h
IER0RESTOREMASK_clk .set 0008h
IER1RESTOREMASK_clk .set 0h
```

CLK\_isr:

```
HWI_enter AR_T_MASK_clk, AC_MASK_clk, MISC1_MASK_clk,
MISC2_MASK_clk, MISC3_MASK_clk, IER0DISABLEMASK_clk,
IER1DISABLEMASK_clk
PRD_tick
HWI_exit AR_T_MASK_clk, AC_MASK_clk, MISC1_MASK_clk,
MISC2_MASK_clk, MISC3_MASK_clk, IER0RESTOREMASK_clk,
IER1RESTOREMASK_clk
```

**Example #2:**

Calling a C function from within an HWI\_enter/HWI\_exit block:

Specify all registers in the C convention class, save-by-caller. Use the appropriate register save masks with the HWI\_enter macro:

```
HWI_enter C55_AR_T_SAVE_BY_CALLER_MASK,  
C55_AC_SAVE_BY_CALLER_MASK,  
C55_MISC1_SAVE_BY_CALLER_MASK,  
C55_MISC2_SAVE_BY_CALLER_MASK,  
C55_MISC3_SAVE_BY_CALLER_MASK, user_ier0_mask,  
user_ier1_mask
```

The HWI\_enter macro

- ❑ preserves the specified set of registers that are being declared as trashable by the called function
- ❑ places the processor status register bit settings as required by C compiler conventions
- ❑ aligns the stack pointers to even address boundaries, as well as remembering any such adjustments made to the SP and SSP registers

The user's C function must have a leading underscore as seen in this example:

```
call _myCfunction;
```

When exiting the hardware interrupt, you need to call HWI\_exit with the following macro:

```
HWI_exit C55_AR_T_SAVE_BY_CALLER_MASK,  
C55_AC_SAVE_BY_CALLER_MASK,  
C55_MISC1_SAVE_BY_CALLER_MASK,  
C55_MISC2_SAVE_BY_CALLER_MASK,  
C55_MISC3_SAVE_BY_CALLER_MASK,          user_ier0_mask,  
user_ier1_mask
```

The HWI\_exit macro restores the CPU state that was originally set by the HWI\_enter macro. It alerts the SWI scheduler to attend to any kernel scheduling activity that is required.

**See Also**

HWI\_exit

**HWI\_exit***Hardware ISR epilog***C Interface**

<b>Syntax</b>	none
<b>Parameters</b>	none
<b>Return Value</b>	none

**Assembly Interface**

<b>Syntax</b>	HWI_exit MASK IMRRESTOREMASK
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE
<b>Postconditions</b>	intrm = 0
<b>Modifies</b>	Restores all registers saved in HWI_enter

**Assembly Interface**

<b>Syntax</b>	HWI_exit AR_T_MASK, AC_MASK, MISC1_MASK, MISC2_MASK, MISC3_MASK, IER0RESTOREMASK, IER1RESTOREMASK
<b>Preconditions</b>	none
<b>Postconditions</b>	intrm=0
<b>Modifies</b>	Restores all registers saved with the HWI_enter mask

**Reentrant**                      yes

**Description**                      HWI\_exit is an API (assembly macro) which is used to restore the context that existed before a DSP/BIOS interrupt service routine (ISR) was invoked.

HWI\_exit is used by ISRs that are user-dispatched, as opposed to ISRs that are handled by the HWI dispatcher. HWI\_exit must not be issued by ISRs that are handled by the HWI dispatcher.

If the HWI dispatcher is not used by an HWI object, HWI\_exit must be the last statement in an ISR that uses DSP/BIOS API calls which could trigger other DSP/BIOS objects, such as posting an SWI or semaphore.





HWI\_exit restores the registers specified by AR\_T\_MASK, AC\_MASK, MISC1\_MASK, MISC2\_MASK, and MISC3\_MASK. These masks are used to specify the set of registers that were saved by HWI\_enter.



HWI\_exit restores the registers specified by MASK. This mask is used to specify the set of registers that were saved by HWI\_enter.

HWI\_enter and HWI\_exit must surround all statements in any DSP/BIOS assembly language ISRs that call C functions only for ISRs that are not dispatched by the HWI dispatcher.

HWI\_exit calls the DSP/BIOS Software Interrupt manager if DSP/BIOS itself is not in the middle of updating critical data structures, or if no currently interrupted ISR is also in a HWI\_enter/ HWI\_exit region. The DSP/BIOS SWI manager services all pending SWI handlers (functions).



Of the interrupts in IMRRESTOREMASK, HWI\_exit only restores those enabled upon entering the ISR. HWI\_exit does not affect the status of interrupt bits that are not in IMRRESTOREMASK. If upon exiting an ISR you do not wish to restore an interrupt that was disabled with HWI\_enter, do not set that interrupt bit in the IMRRESTOREMASK in HWI\_exit.

If upon exiting an ISR you wish to enable an interrupt that was disabled upon entering the ISR, set the corresponding bit in IMR register. (Including a bit in IMR in the IMRRESTOREMASK of HWI\_exit does not enable the interrupt if it was disabled when the ISR was entered.)



Of the interrupts in IER0RESTOREMASK/IER1RESTOREMASK, HWI\_exit only restores those that were disabled upon entering the ISR. HWI\_exit does not affect the status of interrupt bits that are not in IER0RESTOREMASK/IER1RESTOREMASK. If upon exiting an ISR you do not wish to restore one of the interrupts that were disabled with HWI\_enter, do not set that interrupt bit in the IER0[IER1]RESTOREMASK in HWI\_exit.

If upon exiting an ISR you do wish to enable an interrupt that was disabled upon entering the ISR, set the corresponding bit in IER0[IER1]RESTOREMASK before calling HWI\_exit. (Simply setting bits in IER0RESTOREMASK/IER1RESTOREMASK that is passed as argument to HWI\_exit does not result in enabling the corresponding interrupts if those were not originally disabled by the HWI\_enter macro.)

## Constraints and Calling Context

- ☐ This API should not be used for the NMI HWI function.
- ☐ This API must not be called if the HWI object that runs the ISR uses the HWI dispatcher.
- ☐ If the HWI dispatcher is not used, this API must be the last operation in an ISR that uses any DSP/BIOS API calls that might post or affect

a software interrupt or semaphore. The HWI dispatcher can be used instead of this API, allowing the function to be written completely in C and allowing you to reduce code size.

- ❑ On the C54 platform, the MASK parameter must match the corresponding parameter used for HWI\_enter.
- ❑ On the C55 platform, the AR\_T\_MASK, AC\_MASK, MISC1\_MASK, MISC2\_MASK, and MISC3\_MASK parameters must match the corresponding parameters used for HWI\_enter.
- ❑ This API cannot be called from the program's main function.
- ❑ This API cannot be called from an SWI, TSK, or IDL function.
- ❑ This API cannot be called from a CLK function.
- ❑ On the C54x platform, the postconditions of HWI\_enter do not completely satisfy C function calling conventions. Specifically, after calling HWI\_enter, the CPL bit is not set. Before calling a C function, you must set the CPL bit and restore it afterwards to satisfy a precondition for calling HWI\_exit. If using the HWI dispatcher, this is not necessary as the dispatcher assumes a C function and performs this step prior to calling the user function.



### Example



CLK\_isr:

```
HWI_enter C54_CNOTPRESERVED, 0008h
PRD_tick
HWI_exit C54_CNOTPRESERVED, 0008h
```

### Examples



Example #1:

```
.include c55.h55
```

```
AR_T_MASK_clk      .set C55_AR57_MASK
AC_MASK_clk        .set C55_AC3_MASK
MISC1_MASK_clk     .set 0
MISC2_MASK_clk     .set C55_TRN1
MISC3_MASK_clk     .set 0
IER0DISABLEMASK_clk .set 0008h
IER1DISABLEMASK_clk .set 0h
IER0RESTOREMASK_clk .set 0008h
IER1RESTOREMASK_clk .set 0h
```

CLK\_isr:

```
HWI_enter AR_T_MASK_clk, AC_MASK_clk, MISC1_MASK_clk,
MISC2_MASK_clk, MISC3_MASK_clk, IER0DISABLEMASK_clk,
IER1DISABLEMASK_clk
PRD_tick
HWI_exit AR_T_MASK_clk, AC_MASK_clk, MISC1_MASK_clk,
MISC2_MASK_clk, MISC3_MASK_clk, IER0RESTOREMASK_clk,
IER1RESTOREMASK_clk
```

**Example #2:**

Calling a C function from within an HWI\_enter/HWI\_exit:

Specify all registers in the C convention class, save-by-caller. Use the appropriate register save masks with the HWI\_enter macro:

```
HWI_enter C55_AR_T_SAVE_BY_CALLER_MASK,  
C55_AC_SAVE_BY_CALLER_MASK,  
C55_MISC1_SAVE_BY_CALLER_MASK,  
C55_MISC2_SAVE_BY_CALLER_MASK,  
C55_MISC3_SAVE_BY_CALLER_MASK, user_ier0_mask,  
user_ier1_mask
```

The HWI\_enter macro

- ❑ preserves the specified set of registers that are being declared as trashable by the called function
- ❑ places the processor status register bit settings as required by C compiler conventions
- ❑ aligns the stack pointers to even address boundaries, as well as remembering any such adjustments made to the SP and SSP registers

The user's C function must have a leading underscore as seen in this example:

```
call _myCfunction;
```

When exiting the hardware interrupt, you need to call HWI\_exit with the following macro:

```
HWI_exit C55_AR_T_SAVE_BY_CALLER_MASK,  
C55_AC_SAVE_BY_CALLER_MASK,  
C55_MISC1_SAVE_BY_CALLER_MASK,  
C55_MISC2_SAVE_BY_CALLER_MASK,  
C55_MISC3_SAVE_BY_CALLER_MASK, user_ier0_mask,  
user_ier1_mask
```

The HWI\_exit macro restores the CPU state that was originally set by the HWI\_enter macro. It alerts the SWI scheduler to attend to any kernel scheduling activity that is required.

**See Also**

HWI\_enter

**HWI\_restore***Restore global interrupt enable state***C Interface**

<b>Syntax</b>	HWI_restore(oldST1);
<b>Parameters</b>	Uns            oldST1;
<b>Returns</b>	Void

**Assembly Interface**

<b>Syntax</b>	HWI_restore
<b>Preconditions</b>	al = a 16-bit mask intm = 1
<b>Postconditions</b>	intm will be set to the value of bit 11 of al
<b>Modifies</b>	ag, ah, al, intm

**Assembly Interface**

<b>Syntax</b>	HWI_restore
<b>Preconditions</b>	ac0l == mask (intm is set to the value of bit 11) intm = 1
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, intm

<b>Reentrant</b>	no
------------------	----

<b>Description</b>	<p>HWI_restore sets the intm bit in the st1 register using bit 11 of the oldst1 parameter. If bit 11 is 1, the intm bit is not modified. If bit 11 is 0, the intm bit is set to 0, which enables interrupts.</p>
--------------------	--

When you call HWI\_disable, the previous contents of the st1 register are returned. You can use this returned value with HWI\_restore.

A context switch may occur when calling HWI\_restore if HWI\_restore reenables interrupts and if a higher-priority HWI occurred while interrupts were disabled.

### Constraints and Calling Context

- ❑ HWI\_restore cannot be called from the program's main function.

### Example

```
oldST1 = HWI_disable(); /* disable interrupts */
    'do some critical operation'
HWI_restore(oldST1);
    /* re-enable interrupts if they
       were enabled at the start of the
       critical section */
```

### See Also

HWI\_enable  
HWI\_disable

## 2.9 IDL Module

The IDL module is the idle thread manager.

### Functions

- ☐ **IDL\_run.** Make one pass through idle functions.

### Description

The IDL module manages the lowest-level threads in the application. In addition to user-created functions, the IDL module executes DSP/BIOS functions that handle host communication and CPU load calculation.

There are four kinds of threads that can be executed by DSP/BIOS programs: hardware interrupts (HWI module), software interrupts (SWI module), tasks (TSK module), and background threads (IDL module). Background threads have the lowest priority, and execute only if no hardware interrupts, software interrupts, or tasks need to run.

An application's main function must return before any DSP/BIOS threads can run. After the return, DSP/BIOS runs the idle loop. Once an application is in this loop, HWI hardware interrupts, SWI software interrupts, PRD periodic functions, TSK task functions, and IDL background threads are all enabled.

The functions for IDL objects registered with the Configuration Tool are run in sequence each time the idle loop runs. IDL functions are called from the IDL context. IDL functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

When RTA is enabled (see page 2–75), an application contains an IDL\_cpuLoad object, which runs a function that provides data about the CPU utilization of the application. In addition, the LNK\_dataPump function handles host I/O in the background, and the RTA\_dispatch function handles run-time analysis communication.

The IDL Function Manager allows you to insert additional functions that are executed in a loop whenever no other processing (such as hardware ISRs or higher-priority tasks) is required.

### IDL Manager Properties

The following global properties can be set for the IDL module on the IDL Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the IDL objects.
- ☐ **Auto calculate idle loop instruction count.** When this box is checked, the program runs through the IDL functions at system startup to get an approximate value for the idle loop instruction count. This value, saved in the global variable CLK\_D\_idleTime, is read by the host and used in CPU load calculation. By default, the instruction count includes all IDL functions, not just LNK\_dataPump, RTA\_dispatcher, and IDL\_cpuLoad. You can remove an IDL function

from the calculation by removing the checkmark from the Include in CPU load calibration box in an IDL object's Properties dialog.

Remember that functions included in the calibration are run before the main function runs. These functions should not access data structures that are not initialized before the main function runs. In particular, functions that perform any of the following actions should not be included in the idle loop calibration:

- enabling hardware interrupts or the SWI or TSK schedulers
  - using CLK APIs to get the time
  - accessing PIP objects
  - blocking tasks
  - creating dynamic objects
- ☐ **Idle Loop Instruction Count.** This is the number of instruction cycles required to perform the IDL loop and the default IDL functions (LNK\_dataPump, RTA\_dispatcher, and IDL\_cpuLoad) that communicate with the host. Since these functions are performed whenever no other processing is needed, background processing is subtracted from the CPU load before it is displayed.

## IDL Object Properties

Each idle function runs to completion before another idle function can run. It is important, therefore, to ensure that each idle function completes (that is, returns) in a timely manner.

The following properties can be set for an IDL object on the IDL Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this IDL object.
- ☐ **function.** The function to be executed. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- ☐ **Include in CPU load calibration.** You can remove an individual IDL function from the CPU load calculation by removing the checkmark from this box. The CPU load calibration is performed only if the Auto calculate idle loop instruction count box is checked in the IDL Manager Properties. You should remove a function from the calculation if it blocks or depends on variables or structures that are not initialized until the main function runs.

## IDL- Execution Graph Interface

Time spent performing IDL functions is not directly traced. However, the Other Threads row in the Execution Graph, which you can open by choosing DSP/BIOS→Execution Graph, includes time spent performing both HWI and IDL functions.

**IDL\_run***Make one pass through idle functions***C Interface****Syntax** IDL\_run();**Parameters** Void**Return Value** Void**Assembly Interface** none

**Description** IDL\_run makes one pass through the list of configured IDL objects, calling one function after the next. IDL\_run returns after all IDL functions have been executed one time. IDL\_run is not used by most DSP/BIOS applications since the IDL functions are executed in a loop when the application returns from main. IDL\_run is provided to allow easy integration of the real-time analysis features of DSP/BIOS (for example, LOG and STS) into existing applications.

IDL\_run must be called to transfer the real-time analysis data to and from the host computer. Though not required, this is usually done during idle time when no HWI or SWI threads are running.

**Note:**

BIOS\_init and BIOS\_start must be called before IDL\_run to ensure that DSP/BIOS has been initialized. For example, the DSP/BIOS boot file contains the following system calls around the call to main:

```
BIOS_init(); /* initialize DSP/BIOS */
main();
BIOS_start() /* start DSP/BIOS */
IDL_loop(); /* call IDL_run in an infinite loop */
```

**Constraints and Calling Context**

- ❑ IDL\_run cannot be called by an HWI or SWI function.



## 2.10 LCK Module

The LCK module is the resource lock manager.

### Functions

- ☐ **LCK\_create.** Create a resource lock
- ☐ **LCK\_delete.** Delete a resource lock
- ☐ **LCK\_pend.** Acquire ownership of a resource lock
- ☐ **LCK\_post.** Relinquish ownership of a resource lock

### Constants, Types, and Structures

```
typedef struct LCK_Obj *LCK_Handle; /* resource handle */

/* lock object */
typedef struct LCK_Attrs LCK_Attrs;

struct LCK_Attrs {
    Int dummy;
};

LCK_Attrs LCK_ATTRS = {0}; /* default attribute values */
```

### Description

The lock module makes available a set of functions that manipulate lock objects accessed through handles of type `LCK_Handle`. Each lock implicitly corresponds to a shared global resource, and is used to arbitrate access to this resource among several competing tasks.

The LCK module contains a pair of functions for acquiring and relinquishing ownership of resource locks on a per-task basis. These functions are used to bracket sections of code requiring mutually exclusive access to a particular resource.

LCK lock objects are semaphores that potentially cause the current task to suspend execution when acquiring a lock.

### LCK Manager Properties

The following global property can be set for the LCK module on the LCK Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the LCK objects created with the Configuration Tool.

### LCK Object Properties

The following property can be set for a LCK object on the LCK Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this LCK object.

**LCK\_create***Create a resource lock***C Interface**

<b>Syntax</b>	lock = LCK_create(attrs);
<b>Parameters</b>	LCK_Attrs    attrs;    /* pointer to lock attributes */
<b>Return Value</b>	LCK_Handle lock;    /* handle for new lock object */

**Assembly Interface**

none

**Description**

LCK\_create creates a new lock object and returns its handle. The lock has no current owner and its corresponding resource is available for acquisition through LCK\_pend.

If attrs is NULL, the new lock is assigned a default set of attributes. Otherwise the lock's attributes are specified through a structure of type LCK\_Attrs.

**Note:**

At present, no attributes are supported for lock objects.

All default attribute values are contained in the constant LCK\_ATTRS, which can be assigned to a variable of type LCK\_Attrs prior to calling LCK\_create.

LCK\_create calls MEM\_alloc to dynamically create the object's data structure. MEM\_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2–136.

**Constraints and Calling Context**

- ❑ LCK\_create cannot be called from an SWI or HWI.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

LCK\_delete  
LCK\_pend  
LCK\_post

LCK\_delete

Delete a resource lock

C Interface

Syntax	LCK_delete(lock);
Parameters	LCK_Handle lock;       /* lock handle */
Return Value	Void

Assembly Interface       none

Description       LCK\_delete uses MEM\_free to free the lock referenced by lock.

LCK\_delete calls MEM\_free to delete the LCK object. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

- Constraints and Calling Context
- ❑ LCK\_delete cannot be called from an SWI or HWI.
  - ❑ No task should be awaiting ownership of the lock.
  - ❑ No check is performed to prevent LCK\_delete from being used on a statically-created object. If a program attempts to delete a lock object that was created using the Configuration Tool, SYS\_error is called.

See Also       LCK\_create  
LCK\_post  
LCK\_pend

**LCK\_pend***Acquire ownership of a resource lock***C Interface**

<b>Syntax</b>	status = LCK_pend(lock, timeout);
<b>Parameters</b>	LCK_Handle lock;      /* lock handle */ Uns            timeout; /* return after this many system clock ticks */
<b>Return Value</b>	Bool            status;    /* TRUE if successful, FALSE if timeout */

**Assembly Interface**

none

**Description**

LCK\_pend acquires ownership of lock, which grants the current task exclusive access to the corresponding resource. If lock is already owned by another task, LCK\_pend suspends execution of the current task until the resource becomes available.

The task owning lock can call LCK\_pend any number of times without risk of blocking, although relinquishing ownership of the lock requires a balancing number of calls to LCK\_post.

LCK\_pend results in a context switch if this LCK timeout is greater than 0 and the lock is already held by another thread.

LCK\_pend returns TRUE if it successfully acquires ownership of lock, returns FALSE if timeout.

**Constraints and Calling Context**

- ❑ lock must be a handle for a resource lock object created through a prior call to LCK\_create.
- ❑ This function is callable by SWI and HWI threads with timeout equal to zero only.

**See Also**

LCK\_create  
LCK\_delete  
LCK\_post

**LCK\_post***Relinquish ownership of a resource LCK***C Interface**

<b>Syntax</b>	LCK_post(lock);
<b>Parameters</b>	LCK_Handle lock;      /* lock handle */
<b>Return Value</b>	Void

**Assembly Interface**      none

**Description**      LCK\_post relinquishes ownership of lock, and resumes execution of the first task (if any) awaiting availability of the corresponding resource. If the current task calls LCK\_pend more than once with lock, ownership remains with the current task until LCK\_post is called an equal number of times.

LCK\_post results in a context switch if a higher priority thread is currently pending on the lock.

**Constraints and  
Calling Context**

- ❑ lock must be a handle for a resource lock object created through a prior call to LCK\_create.
- ❑ When called within an HWI, the code sequence calling LCK\_post must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**See Also**

LCK\_create  
LCK\_delete  
LCK\_pend

## 2.11 LOG Module

The LOG module captures events in real time.

### Functions

- ☐ LOG\_disable. Disable the system log.
- ☐ LOG\_enable. Enable the system log.
- ☐ LOG\_error. Write a user error event to the system log.
- ☐ LOG\_event. Append unformatted message to message log.
- ☐ LOG\_message. Write a user message event to the system log.
- ☐ LOG\_printf. Append formatted message to message log.
- ☐ LOG\_reset. Reset the system log.

### Description

The Event Log is used to capture events in real time while the target program executes. You can use the system log, or create user-defined logs. If the logtype is circular, the log buffer of size buflen contains the last buflen elements. If the logtype is fixed, the log buffer contains the first buflen elements.

The system log stores messages about system events for the types of log tracing you have enabled. See the TRC Module, page 2–312, for a list of events that can be traced in the system log.

You can add messages to user logs or the system log by using LOG\_printf or LOG\_event. To reduce execution time, log data is always formatted on the host.

LOG\_error writes a user error event to the system log. This operation is not affected by any TRC trace bits; an error event is always written to the system log. LOG\_message writes a user message event to the system log, provided that both TRC\_GBLHOST and TRC\_GBLTARG (the host and target trace bits, respectively) traces are enabled.

When a problem is detected on the target, it is valuable to put a message in the system log. This allows you to correlate the occurrence of the detected event with the other system events in time. LOG\_error and LOG\_message can be used for this purpose.

Log buffers are of a fixed size and reside in data memory. Individual messages use four words of storage in the log's buffer. The first word holds a sequence number that allows the Event Log to display logs in the correct order. The remaining three words contain data specified by the call that wrote the message to the log.

Log buffers are of a fixed size and reside in data memory. Individual messages hold four elements in the log's buffer. The first element holds



a sequence number that allows the Event Log to display logs in the correct order. The remaining three elements contain data specified by the call that wrote the message to the log.

Each log event buffer uses four words in the small memory model and eight words in the large memory model.

See the *Code Composer Studio* online tutorial for examples of how to use the LOG Manager.

## LOG Manager Properties

The following global property can be set for the LOG module on the LOG Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the LOG objects.

## LOG Object Properties

The following properties can be set for a log object on the LOG Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this LOG object.
- ☐ **bufseg.** The name of a memory segment to contain the log buffer.
- ☐ **buflen.** The length of the log buffer (in words).
- ☐ **logtype.** The type of the log: circular or fixed. Events added to a full circular log overwrite the oldest event in the buffer, whereas events added to a full fixed log are dropped.
  - **Fixed.** The log stores the first messages it receives and stops accepting messages when its message buffer is full.
  - **Circular.** The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.

- ☐ **datatype.** Choose printf if you use LOG\_printf to write to this log and provide a format string.

Choose raw data if you want to use LOG\_event to write to this log and have the Event Log apply a printf-style format string to all records in the log.

- ☐ **format.** If you choose raw data as the datatype, type a printf-style format string in this field. Provide up to three (3) conversion characters (such as %d) to format words two, three, and four in all records in the log. Do not put quotes around the format string. The format string can use %d, %x, %o, %s, %r, and %p conversion characters; it cannot use other types of conversion characters.

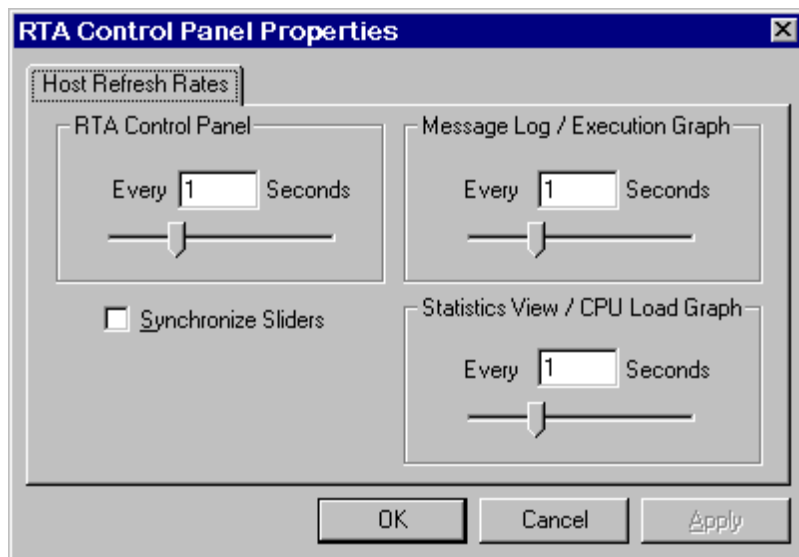
See LOG\_printf, page 2–122, and LOG\_event, page 2–120, for information about the structure of a log record.

**LOG - Code Composer  
Studio Interface**

You can view log messages in real time while your program is running with the Event Log. A pull-down menu provides a list of the logs you can view. To see the system log as a graph, choose DSP/BIOS→ Execution Graph Details. To see a user log, choose DSP/BIOS→Event Log and select the log or logs you want to see. The Property Page for the Message Log allows you to select a file to which the log messages will be written. Right-click on the Message Log and select Property Page to name this file. You cannot open the named log file until you close the Message Log window.

You can also control how frequently the host polls the target for log information. Right-click on the RTA Control Panel and choose the Property Page to set the refresh rate as shown in Figure 2-1. If you set the refresh rate to 0, the host does not poll the target unless you right-click on the log window and choose Refresh Window from the pop-up menu

Figure 2-1. RTA Control Panel Properties Page





LOG\_disable

Disable a message log

C Interface

Syntax	LOG_disable(log);
Parameters	LOG_Handle log;      /* log object handle */
Return Value	Void

Assembly Interface



Syntax	LOG_disable
Preconditions	ar2 = address of the LOG object
Postconditions	none
Modifies	c

Assembly Interface



Syntax	LOG_disable
Preconditions	xar0 = address of the LOG object
Postconditions	none
Modifies	none

Reentrant	no
-----------	----

Description	LOG_disable disables the logging mechanism and prevents the log buffer from being modified.
-------------	---

Example	LOG_disable(&trace);
---------	----------------------

See Also	LOG_enable LOG_reset
----------	-------------------------

**LOG\_enable***Enable a message log***C Interface**

<b>Syntax</b>	LOG_enable(log);
<b>Parameters</b>	LOG_Handle log;      /* log object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	LOG_enable
<b>Preconditions</b>	ar2 = address of the LOG object
<b>Postconditions</b>	none
<b>Modifies</b>	c

**Assembly Interface**

<b>Syntax</b>	LOG_enable
<b>Preconditions</b>	xar0 = address of the LOG object
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	no
<b>Description</b>	LOG_enable enables the logging mechanism and allows the log buffer to be modified.
<b>Example</b>	LOG_enable(&trace);
<b>See Also</b>	LOG_disable LOG_reset

**LOG\_error**

*Write an error message to the system log*

**C Interface**

<b>Syntax</b>	LOG_error(format, arg0);
<b>Parameters</b>	String        format;    /* printf-style format string */ Arg           arg0;      /* copied to second word of log record */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	LOG_error format [section]
<b>Preconditions</b>	ar2 = format bh = arg0 dp = GBL_A_SYSPAGE
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, bl, c, t, tc

**Assembly Interface**



<b>Syntax</b>	LOG_error format [section]
<b>Preconditions</b>	format and optional section arguments are directly passed as macro parameters xar1 = arg0
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, tc1, tc2, ac0

**Reentrant**                yes

**Description**            LOG\_error writes a program-supplied error message to the system log, which is defined in the default configuration by the LOG\_system object. LOG\_error is not affected by any TRC bits; an error event is always written to the system log.

The format argument can contain any of the conversion characters supported for LOG\_printf. See LOG\_printf for details.

The LOG\_error assembly macro takes an optional section argument. If you omit this argument, assembly code following the macro is assembled into the .text section. If you want your program to be assembled into another section, specify another section name when calling the macro.

**Example**

```
Void UTL_doError(String s, Int errno)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = '%s'", s);
}
```

**See Also**

LOG\_event  
LOG\_message  
LOG\_printf  
TRC\_disable  
TRC\_enable

**LOG\_message**

*Write a program-supplied message to the system log*

**C Interface**

<b>Syntax</b>	LOG_message(format, arg0);
<b>Parameters</b>	String        format;    /* printf-style format string */ Arg           arg0;      /* copied to second word of log record */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	LOG_message format [section]
<b>Preconditions</b>	ar2 = format bh = arg0 dp = GBL_A_SYSPAGE
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, bl, c, t, tc

**Assembly Interface**



<b>Syntax</b>	LOG_message format [section]
<b>Preconditions</b>	format and optional section arguments are directly passed as macro parameters xar1 = arg0
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, tc1, tc2, ac0
<b>Reentrant</b>	yes

**Description**        LOG\_message writes a program-supplied message to the system log, provided that both the host and target trace bits are enabled.

The format argument passed to LOG\_message can contain any of the conversion characters supported for LOG\_printf. See LOG\_printf, page 2-122, for details.

The LOG\_message assembly macro takes an optional section argument. If you do not specify a section argument, assembly code following the macro is assembled into the .text section by default. If you do not want your program to be assembled into the .text section, you should specify the desired section name when calling the macro.

**Example**

```
Void UTL_doMessage(String s, Int errno)
{
    LOG_message("SYS_error called: error id = 0x%x", errno);
    LOG_message("SYS_error called: string = '%s'", s);
}
```

**See Also**

LOG\_error  
LOG\_event  
LOG\_printf  
TRC\_disable  
TRC\_enable

**LOG\_event**

*Append an unformatted message to a message log*

**C Interface**

<b>Syntax</b>	LOG_event(log, arg0, arg1, arg2);
<b>Parameters</b>	LOG_Handle log;       /* log objecthandle */ Arg       arg0;       /* copied to second word of log record */ Arg       arg1;       /* copied to third word of log record */ Arg       arg2;       /* copied to fourth word of log record */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	LOG_event
<b>Preconditions</b>	ar2 = address of the LOG object bh = arg0 bl = arg1 t = arg2
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, c, tc

**Assembly Interface**



<b>Syntax</b>	LOG_message format [section]
<b>Preconditions</b>	xar0 = address of the LOG object xar1 = arg0 xar2 = arg1 xar3 = arg2
<b>Postconditions</b>	none
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, tc1, tc2, ac0
<b>Reentrant</b>	yes

**Description** LOG\_event copies a sequence number and three arguments to the specified log buffer. Each log message uses four words in the small memory model and eight words in the large memory model. The contents of the four words written by LOG\_event are shown here:

LOG_event	Sequence #	arg0	arg1	arg2
-----------	------------	------	------	------

You can format the log by using LOG\_printf instead of LOG\_event.

If you want the Event Log to apply the same printf-style format string to all records in the log, use the Configuration Tool to choose raw data for the datatype property of this log object and typing a format string for the format property.

If the logtype is circular, the log buffer of size buflen contains the last buflen elements. If the logtype is fixed, the log buffer contains the first buflen elements.

Any combination of threads can write to the same log. Internally, hardware interrupts are temporarily disabled during a call to LOG\_event. Log messages are never lost due to thread preemption.

**Example**

```
LOG_event(&trace, (Arg)value1, (Arg)value2,CLK_gettime);
```

**See Also** LOG\_error  
LOG\_printf  
TRC\_disable  
TRC\_enable



LOG\_printf

Append a formatted message to a message log

C Interface

Syntax	LOG_printf(log, format); or LOG_printf(log, format,arg0); or LOG_printf(log, format, arg0, arg1);
Parameters	LOG_Handle log; /* log object handle */ String format; /* printf format string */ Arg arg0; /* value for first format string token */ Arg arg1; /* value for second format string token */
Return Value	Void

Assembly Interface



Syntax	LOG_printf format [section]
Preconditions	ar2 = address of the LOG object bh = arg0 bl = arg1
Postconditions	none
Modifies	ag, ah, al, ar0, ar2, ar3, c, t, tc

Assembly Interface



Syntax	LOG_printf format [section]
Preconditions	xar0 = address of the LOG object xar1 = arg0 xar2 = arg1
Postconditions	none
Modifies	xar0, xar1, xar2, xar3, xar4, t0, tc1, tc2, ac0
Reentrant	yes

**Description**

As a convenience for C (as well as assembly language) programmers, the LOG module provides a variation of the ever-popular printf. LOG\_printf copies a sequence number, the format address, and two arguments to the specified log buffer.

To reduce execution time, log data is always formatted on the host. The format string is stored on the host and accessed by the Event Log.

The arguments passed to LOG\_printf must be integers, strings, or a pointer if the special %r conversion character is used. The format string can use any of the conversion characters found in Table 2-3.

*Table 2-3. Conversion Characters for LOG\_printf*

Conversion Character	Description
%d	Signed integer
%x	Unsigned hexadecimal integer
%o	Unsigned octal integer
%s	<p>Character string</p> <p>This character can only be used with constant string pointers. That is, the string must appear in the source and be passed to LOG_printf. For example, the following is supported:</p> <pre>char *msg = "Hello world!"; LOG_printf(&amp;trace, "%s", msg);</pre> <p>However, the following example is not supported:</p> <pre>char msg[100]; strcpy(msg, "Hello world!"); LOG_printf(&amp;trace, "%s", msg);</pre> <p>If the string appears in the COFF file and a pointer to the string is passed to LOG_printf, then the string in the COFF file is used by the Event Log to generate the output. If the string can not be found in the COFF file, the format string is replaced with <b>*** ERROR: 0x%x 0x%x ***\n</b>, which displays all arguments in hexadecimal.</p>

Conversion Character	Description
%r	<p>Symbol from symbol table</p> <p>This is an extension of the standard printf format tokens. This character treats its parameter as a pointer to be looked up in the symbol table of the executable and displayed. That is, %r displays the symbol (defined in the executable) whose value matches the value passed to %r. For example:</p> <pre>Int testval = 17; LOG_printf("%r = %d", &amp;testval, testval);</pre> <p>displays:</p> <pre>testval = 17</pre> <p>If no symbol is found for the value passed to %r, the Event Log uses the string &lt;unknown symbol&gt;.</p>
%p	data pointer

If you want the Event Log to apply the same printf-style format string to all records in the log, use the Configuration Tool to choose raw data for the datatype property of this log object and typing a format string for the format property.

The LOG\_printf assembly macro takes an optional section parameter. If you do not specify a section parameter, assembly code following the LOG\_printf macro is assembled into the .text section by default. If you do not want your program to be assembled into the .text section, you should specify the desired section name as the second parameter to the LOG\_printf call.

Each log message uses four words in the small memory model and eight words in the large memory model. The contents of the four words written by LOG\_printf are shown here:

LOG_printf	Sequence #	arg0	arg1	Format address
------------	------------	------	------	----------------

You configure the characteristics of a log in the Configuration Tool. If the logtype is circular, the log buffer of size buflen contains the last buflen elements. If the logtype is fixed, the log buffer contains the first buflen elements.

Any combination of threads can write to the same log. Internally, hardware interrupts are temporarily disabled during a call to LOG\_printf. Log messages are never lost due to thread preemption.

**Constraints and  
Calling Context**

LOG\_printf (even the C version) supports 0, 1, or 2 arguments after the format string.

**Example**

```
LOG_printf(&trace, "hello world");  
LOG_printf(&trace, "Size of Int is: %d", sizeof(Int));
```

**See Also**

LOG\_error  
LOG\_event  
TRC\_disable  
TRC\_enable

**LOG\_reset**

*Reset a message log*

**C Interface**

<b>Syntax</b>	LOG_reset(log);
<b>Parameters</b>	LOG_Handle   log   /* log object handle */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	LOG_reset
<b>Preconditions</b>	ar2 = address of the LOG object
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar3, ar4, c

**Assembly Interface**



<b>Syntax</b>	LOG_reset
<b>Preconditions</b>	xar0 = address of the LOG object
<b>Postconditions</b>	none
<b>Modifies</b>	xar0, xar1, xar2, t0, ac0

**Reentrant**                      no

**Description**                      LOG\_reset enables the logging mechanism and allows the log buffer to be modified starting from the beginning of the buffer, with sequence number starting from 0.

LOG\_reset does not disable interrupts or otherwise protect the log from being modified by an HWI or other thread. It is therefore possible for the log to contain inconsistent data if LOG\_reset is preempted by an HWI or other thread that uses the same log.

**Example**                              LOG\_reset(&trace);

**See Also**                              LOG\_disable  
LOG\_enable

## 2.12 MBX Module

The MBX module is the mailbox manager.

### Functions

- ☐ **MBX\_create.** Create a mailbox
- ☐ **MBX\_delete.** Delete a mailbox
- ☐ **MBX\_pend.** Wait for a message from mailbox
- ☐ **MBX\_post.** Post a message to mailbox

### Constants, Types, and Structures

```
typedef struct MBX_Obj *MBX_Handle;
/* handle for mailbox object */

struct MBX_Attrs {          /* mailbox attributes */
    Int      segid;
};

MBX_Attrs MBX_ATTRS = { /* default attribute values */
    0,
};
```

### Description

The MBX module makes available a set of functions that manipulate mailbox objects accessed through handles of type `MBX_Handle`. Mailboxes can hold up to the number of messages specified by the Mailbox Length property in the Configuration Tool.

`MBX_pend` is used to wait for a message from a mailbox. The timeout parameter to `MBX_pend` allows the task to wait until a timeout. A timeout value of `SYS_FOREVER` causes the calling task to wait indefinitely for a message. A timeout value of zero (0) causes `MBX_pend` to return immediately. `MBX_pend`'s return value indicates whether the mailbox was signaled successfully.

`MBX_post` is used to send a message to a mailbox. The timeout parameter to `MBX_post` specifies the amount of time the calling task waits if the mailbox is full. If a task is waiting at the mailbox, `MBX_post` removes the task from the queue and puts it on the ready queue. If no task is waiting and the mailbox is not full, `MBX_post` simply deposits the message and returns.

### MBX Manager Properties

The following global property can be set for the MBX module on the MBX Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the MBX objects created with the Configuration Tool.

## MBX Object Properties

The following properties can be set for an MBX object on the MBX Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this MBX object.
- ☐ **Message Size.** The size (in MADUs) of the messages this mailbox can contain.
- ☐ **Mailbox Length.** The number of messages this mailbox can contain.
- ☐ **Element memory segment.** The memory segment to contain the mailbox data buffers.

## MBX Code Composer Studio Interface

The MBX tab of the Kernel/Object View shows information about mailbox objects.

**MBX\_create***Create a mailbox***C Interface**

**Syntax**                    `mbx = MBX_create(msgsize, mbxlength, attrs);`

**Parameters**            `Uns            msgsize; /* size of message */`  
                              `Uns            mbxlength; /* length of mailbox */`  
                              `MBX_Attrs    *attrs;     /* pointer to mailbox attributes */`

**Return Value**           `MBX_Handle mbx;        /* mailbox object handle */`

**Assembly Interface**

none

**Description**

MBX\_create creates a mailbox object which is initialized to contain up to mbxlength messages of size msgsize. If successful, MBX\_create returns the handle of the new mailbox object. If unsuccessful, MBX\_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS\_error, and SYS\_error causes an abort).

If attrs is NULL, the new mailbox is assigned a default set of attributes. Otherwise, the mailbox's attributes are specified through a structure of type MBX\_Attrs.

All default attribute values are contained in the constant MBX\_ATTRS, which can be assigned to a variable of type MBX\_Attrs prior to calling MBX\_create.

MBX\_create calls MEM\_alloc to dynamically create the object's data structure. MEM\_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2–136.

**Constraints and Calling Context**

- ❑ MBX\_create cannot be called from an SWI or HWI.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MBX\_delete  
 SYS\_error



**MBX\_delete**

Delete a mailbox

C Interface

Syntax	MBX_delete(mbx);
Parameters	MBX_Handle mbx;      /* mailbox object handle */
Return Value	Void

Assembly Interface      none

Description      MBX\_delete frees the mailbox object referenced by mbx.

MBX\_delete calls MEM\_free to delete the MBX object. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

- Constraints and Calling Context
- ❑ No tasks should be pending on mbx when MBX\_delete is called.
  - ❑ MBX\_delete cannot be called from an SWI or HWI.
  - ❑ No check is performed to prevent MBX\_delete from being used on a statically-created object. If a program attempts to delete a mailbox object that was created using the Configuration Tool, SYS\_error is called.

See Also      MBX\_create

**MBX\_pend**

*Wait for a message from mailbox*

**C Interface**

<b>Syntax</b>	status = MBX_pend(mbx, msg, timeout);
<b>Parameters</b>	MBX_Handle mbx;     /* mailbox object handle */ Ptr           msg;     /* message pointer */ Uns           timeout; /* return after this many system clock ticks */
<b>Return Value</b>	Bool           status;     /* TRUE if successful, FALSE if timeout */

**Assembly Interface**     none

**Description**     If the mailbox is not empty, MBX\_pend copies the first message into msg and returns TRUE. Otherwise, MBX\_pend suspends the execution of the current task until MBX\_post is called or the timeout expires. The actual time of task suspension can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout is SYS\_FOREVER, the task remains suspended until MBX\_post is called on this mailbox. If timeout is 0, MBX\_pend returns immediately.

If timeout expires (or timeout is 0) before the mailbox is available, MBX\_pend returns FALSE. Otherwise MBX\_pend returns TRUE.

A task switch occurs when calling MBX\_pend if the mailbox is empty and timeout is not 0, or if a higher priority task is blocked on MBX\_post.

- Constraints and Calling Context**

  - ❑ MBX\_pend can only be called from an HWI or SWI if timeout is 0.
  - ❑ MBX\_pend can only be called from within a TSK\_disable / TSK\_enable block if the timeout is 0.
  - ❑ MBX\_pend cannot be called from the program's main function.

**See Also**     MBX\_post

**MBX\_post**

*Post a message to mailbox*

**C Interface**

<b>Syntax</b>	status = MBX_post(mbx, msg, timeout);
<b>Parameters</b>	<div>MBX_Handle mbx       /* mailbox object handle */</div> <div>Ptr           msg;     /* message pointer */</div> <div>Uns           timeout; /* return after this many system clock ticks */</div>
<b>Return Value</b>	Bool           status;   /* TRUE if successful, FALSE if timeout */
<b>Assembly Interface</b>	none

**Description** MBX\_post checks to see if there are any free message slots before copying msg into the mailbox. MBX\_post readies the first task (if any) waiting on mbx.

If the mailbox is full and timeout is SYS\_FOREVER, the task remains suspended until MBX\_pend is called on this mailbox. If timeout is 0, MBX\_post returns immediately. Otherwise, the task is suspended for timeout system clock ticks. The actual time of task suspension can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout expires (or timeout is 0) before the mailbox is available, MBX\_post returns FALSE. Otherwise MBX\_post returns TRUE.

A task switch occurs when calling MBX\_post if a higher priority task is made ready to run, or if there are no free message slots and timeout is not 0.

- Constraints and Calling Context**
- ☐ MBX\_post can only be called from within a TSK\_disable / TSK\_enable block if the timeout is 0.
  - ☐ This function is callable by SWI and HWI threads with timeout equal to zero only.
  - ☐ MBX\_post can be called from the program's main function. However, the number of calls should not be greater than the number of messages the mailbox can hold. Additional calls have no effect.

**See Also**

MBX\_pend

## 2.13 MEM Module

The MEM module is the memory segment manager.

### Functions

- ❑ `MEM_alloc`. Allocate from a memory segment.
- ❑ `MEM_calloc`. Allocate and initialize to 0.
- ❑ `MEM_define`. Define a new memory segment.
- ❑ `MEM_free`. Free a block of memory.
- ❑ `MEM_redefine`. Redefine an existing memory segment.
- ❑ `MEM_stat`. Return the status of a memory segment.
- ❑ `MEM_valloc`. Allocate and initialize to a value.

### Constants, Types, and Structures

```
MEM->MALLOCSEG = 0;      /* segid for malloc, free */

#define MEM_HEADERSIZE /* free block header size */
#define MEM_HEADERMASK /* mask to align on
                        MEM_HEADERSIZE */
#define MEM_ILLEGAL    /* illegal memory address */

MEM_Attrs MEM_ATTRS = { /* default attribute values */
    0
};

typedef struct MEM_Segment {
    Ptr    base;      /* base of the segment */
    Uns    length;    /* size of the segment */
    Uns    space;     /* memory space */
} MEM_Segment;

typedef struct MEM_Stat {
    Uns    size;      /* original size of segment */
    Uns    used;      /* MADUs used in segment */
    Uns    length;    /* largest contiguous block length */
} MEM_Stat;
```

### Description

The MEM module provides a set of functions used to allocate storage from one or more disjointed segments of memory. These memory segments are specified with the Configuration Tool.

MEM always allocates an even number of MADUs and always aligns buffers on an even boundary. This behavior is used to insure that free buffers are always at least two MADUs in length. This behavior does not preclude you from allocating two 512 buffers from a 1K region of on-device memory, for example. It does, however, mean that odd allocations consume one more MADU than expected.

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. To do this, put a checkmark in the No Dynamic Memory Heaps box in the Properties dialog for the MEM manager. If you remove this capability, your program cannot call any of the MEM functions or any object creation functions (such as TSK\_create). You will need to create all objects that will be used by your program with the Configuration Tool. You can also use the Configuration Tool to create or remove the dynamic memory heap from an individual memory segment.

Software modules in DSP/BIOS that allocate storage at run-time use MEM functions; DSP/BIOS does not use the standard C function malloc. DSP/BIOS modules use MEM to allocate storage in the segment selected for that module with the Configuration Tool.

The MEM Manager property, Segment for malloc()/free(), is used to implement the standard C malloc, free, and calloc functions. These functions actually use the MEM functions (with segid = Segment for malloc/free) to allocate and free memory.

---

**Note:**

The MEM module does not set or configure hardware registers associated with a DSP's memory subsystem. Such configuration is the responsibility of the user and is typically handled by software loading programs, or in the case of Code Composer Studio, the startup or menu options. For example, to access external memory on a c6000 platform, the External Memory Interface (EMIF) registers must first be set appropriately before any access. The earliest opportunity for EMIF initialization within DSP/BIOS would be during the user initialization hook (see *Global Settings* in the *API Reference Guide*).

---

## **MEM Manager Properties**

The DSP/BIOS Memory Section Manager allows you to specify the memory segments required to locate the various code and data sections of a DSP/BIOS application.

Note that settings you specify in the Visual Linker normally override settings you specify in the DSP/BIOS Configuration Tool. See the Visual Linker help for details on using the Visual Linker with DSP/BIOS.

The following global properties can be set for the MEM module on the MEM Manager Properties dialog in the Configuration Tool:

### **General tab**

- ☐ **Reuse Startup Code Space.** If this box is checked, the startup code section (.sysinit) can be reused after startup is complete.

- ☐ **Argument Buffer Size.** The size of the .args section. The .args section contains the argc, argv, and envp arguments to the program's main function. Code Composer loads arguments for the main function into the .args section. The .args section is parsed by the boot file.
- ☐ **Stack Size.** The size of the global stack (data stack for the C55x platform) in MADUs. The upper-left corner of the Configuration Tool window shows the estimated minimum global stack size required for this application (as a decimal number).

This size is shown as a hex value in Minimum Addressable Data Units (MADUs). An MADU is the smallest unit of data storage that can be read or written by the CPU. For the c5000 this is a 16-bit word.



- ☐ **System Stack Size (MADUs).** The size of the system stack in MADUs, applicable only on the C55x device.
- ☐ **No Dynamic Memory Heaps.** Put a checkmark in this box to completely disable the ability to dynamically allocate memory and the ability to dynamically create and delete objects. If this box is checked, your program may not call the MEM\_alloc, MEM\_valloc, MEM\_calloc, and malloc or the XXX\_create function for any DSP/BIOS module. If this box is checked, the Segment For DSP/BIOS Objects, Segment for malloc()/free(), and Stack segment for dynamic tasks properties are set to MEM\_NULL.

## BIOS Data tab



- ☐ **Argument Buffer Section (.args).** The memory segment containing the .args section.
- ☐ **Stack Section (.stack).** The memory segment containing the global stack (data stack for the C55x platform). This segment should be located in RAM. The platform architecture requires that both the user and system stacks (pointed to by the XSP and XSSP registers, respectively) reside in the same 64k page of memory, that is, the upper 7 bits of the stack address (SPH) are shared.
- ☐ **System Stack Section (.sysstack).** The memory segment containing the system stack, applicable only on the C55x device.
- ☐ **DSP/BIOS Init Tables (.gblinit).** The memory segment containing the DSP/BIOS global initialization tables.
- ☐ **TRC Initial Value (.trcdata).** The memory segment containing the TRC mask variable and its initial value. This segment must be placed in RAM.
- ☐ **DSP/BIOS Kernel State (.sysdata).** The memory segment containing system data about the DSP/BIOS kernel state.

## BIOS Code tab



## Compiler Sections tab

- ☐ **DSP/BIOS Conf Sections (.obj).** The memory segment containing configuration properties that can be read by the target program.
- ☐ **Segment For DSP/BIOS Objects.** The default memory segment that will contain objects created at run-time with an XXX\_create function. The XXX\_Attrs structure passed to the XXX\_create function can override this default. If you select MEM\_NULL for this property, creation of DSP/BIOS objects at run-time via the XXX\_create functions is disabled.
- ☐ **Segment For malloc() / free().** The memory segment from which space is allocated when a program calls malloc and from which space is freed when a program calls free. If you select MEM\_NULL for this property, dynamic memory allocation at run-time is disabled.
- ☐ **BIOS Code Section (.bios).** The memory segment containing the DSP/BIOS code.
- ☐ **BIOS NORPTB Section (.bios:.norptb).** The memory segment containing DSP/BIOS code that must be placed on the overlay page when the far model is used. This property is visible only if the Function Call Model property in the Global Settings dialog is set to far. This section must be placed in program memory between the addresses 0x0 and 0x7fff. See the application note on DSP/BIOS and TMS320C54x Extended Addressing for more details.
- ☐ **Startup Code Section (.sysinit).** The memory segment containing DSP/BIOS startup initialization code; this memory can be reused after main starts executing.
- ☐ **Function Stub Memory (.hwi).** The memory segment containing dispatch code for interrupt service routines that are configured to be monitored.
- ☐ **Interrupt Service Table Memory (.hwi\_vec).** The memory segment containing the Interrupt Service Table (IST).
- ☐ **RTDX Text Segment (.rtdx\_text).** The memory segment containing the code sections for the RTDX module.
- ☐ **User .cmd File For Non-DSP/BIOS Sections.** Put a checkmark in this box if you want to have full control over the memory used for the sections that follow. You will need to create your own linker command file that begins by including the linker command file created by the Configuration Tool. Your linker command file should then assign memory for the items normally handled by the following properties. See the *TMS320C54x Optimizing Compiler User's Guide* for more details.

- ☐ **Text Section (.text).** The memory segment containing the executable code, string literals, and compiler-generated constants. This segment can be located in ROM or RAM.
- ☐ **Switch Jump Tables (.switch).** The memory segment containing the jump tables for switch statements. This segment can be located in ROM or RAM.
- ☐ **C Variables Section (.bss).** The memory segment containing global and static C variables. At boot or load time, the data in the .cinit section is copied to this segment. This segment should be located in RAM.
- ☐ **Data Initialization Section (.cinit).** The memory segment containing tables for explicitly initialized global and static variables and constants. This segment can be located in ROM or RAM.
- ☐ **C Function Initialization Table (.pinit).** The memory segment containing the table of global object constructors. Global constructors must be called during program initialization. The C/C++ compiler produces a table of constructors to be called at startup. The table is contained in a named section called .pinit. The constructors are invoked in the order that they occur in the table. This segment can be located in ROM or RAM.
- ☐ **Constant Section (.const).** The memory segment containing string constants and data defined with the const C qualifier. If the C compiler is not used, this parameter is unused. This segment can be located in ROM or RAM.
- ☐ **Data Section (.data).** This memory segment contains program data. This segment can be located in ROM or RAM.
- ☐ **Data Section (.cio).** This memory segment contains C standard I/O buffers.

#### Load Address tab

- ☐ **Specify Separate Load Addresses.** If you put a checkmark in this box, you can select separate load addresses for the sections listed on this tab.

Load addresses are useful when, for example, your code must be loaded into ROM, but would run faster in RAM. The linker allows you to allocate sections twice: once to set a load address and again to set a run address.

If you do not select a separate load address for a section, the section loads and runs at the same address.

If you do select a separate load address, the section is allocated as if it were two separate sections of the same size. The load address is where raw data for the section is placed. References to items in the



section refer to the run address. The application must copy the section from its load address to its run address. For details, see the topics on Runtime Relocation and the .label Directive in the Code Generation Tools help or manual.

- ❑ **Load Address - BIOS Code Section (.bios).** The memory segment containing the load allocation of the section that contains DSP/BIOS code.
- ❑ **Load Address - Startup Code Section (.sysinit).** The memory segment containing the load allocation of the section that contains DSP/BIOS startup initialization code.
- ❑ **Load Address - DSP/BIOS Init Tables (.gblinit).** The memory segment containing the load allocation of the section that contains the DSP/BIOS global initialization tables.
- ❑ **Load Address - TRC Initial Value (.trcdata).** The memory segment containing the load allocation of the section that contains the TRC mask variable and its initial value.
- ❑ **Load Address - Text Section (.text).** The memory segment containing the load allocation of the section that contains the executable code, string literals, and compiler-generated constants.
- ❑ **Load Address - Switch Jump Tables (.switch).** The memory segment containing the load allocation of the section that contains the jump tables for switch statements.
- ❑ **Load Address - Data Initialization Section (.cinit).** The memory segment containing the load allocation of the section that contains tables for explicitly initialized global and static variables and constants.
- ❑ **Load Address - C Function Initialization Table (.pinit).** The memory segment containing the load allocation of the section that contains the table of global object constructors.
- ❑ **Load Address - Constant Section (.const).** The memory segment containing the load allocation of the section that contains string constants and data defined with the const C qualifier.
- ❑ **Load Address - Function Stub Memory (.hwi).** The memory segment containing the load allocation of the section that contains dispatch code for interrupt service routines configured to be monitored.
- ❑ **Load Address - Interrupt Service Table Memory (.hwi\_vec).** The memory segment containing the load allocation of the section that contains the Interrupt Service Table.



## MEM Object Properties

- ☐ **Load Address - RTDX Text Segment (.rtdx\_text).** The memory segment containing the load allocation of the section that contains the code sections for the RTDX module.

A memory segment represents a contiguous length of code or data memory in the address space of the processor.

Note that settings you specify in the Visual Linker normally override settings you specify in the DSP/BIOS Configuration Tool. See the Visual Linker help for details on using the Visual Linker with DSP/BIOS.

The following properties can be set for MEM objects on the MEM Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this MEM object.
- ☐ **base.** The address at which this memory segment begins. This value is shown in hex.
- ☐ **len.** The length of this memory segment in MADUs. This value is shown in hex.
- ☐ **create a heap in this memory.** If this box is checked, a heap is created in this memory segment. Memory can be allocated dynamically from a heap. In order to remove the heap from a memory segment, you can select another memory segment that contains a heap for properties that dynamically allocate memory in this memory segment. The properties you should check are in the Memory Section Manager (the Segment for DSP/BIOS objects and Segment for malloc/free properties) and the Task Manager (the Default stack segment for dynamic tasks property). If you disable dynamic memory allocation in the Memory Section Manager, you cannot create a heap in any memory segment.
- ☐ **heap size.** The size of the heap in MADUs to be created in this memory segment.
- ☐ **enter a user defined heap identifier.** If this box is checked, you can define your own identifier label for this heap.
- ☐ **heap identifier label.** If the box above is checked, type a name for this segment's heap.
- ☐ **space.** Type of memory segment. This is set to code for memory segments that store programs, and data for memory segments that store program data.

The predefined memory segments in a configuration file, particularly those for external memory, are dependent on the board template you select. In general, Table lists segments that can be defined for the c5000:

Table 2-4. Typical Memory Segments

Name	Memory segment Type
USERREGS	User scratchpad memory
BIOSREGS	Scratchpad memory reserved for use by DSP/BIOS
VECT	Interrupt vector table
IDATA	Internal data RAM
IPROG	Internal program RAM
EDATA	External data memory
EPROG	External program memory

**MEM Code Composer Studio Interface**

The MEM tab of the Kernel/Object View shows information about memory segments.

**MEM\_alloc***Allocate from a memory segment***C Interface**

<b>Syntax</b>	<code>addr = MEM_alloc(segid, size, align);</code>		
<b>Parameters</b>	<code>Int</code>	<code>segid;</code>	<code>/* memory segment identifier */</code>
	<code>Uns</code>	<code>size;</code>	<code>/* block size in MADUs */</code>
	<code>Uns</code>	<code>align;</code>	<code>/* block alignment */</code>
<b>Return Value</b>	<code>Void</code>	<code>*addr;</code>	<code>/* address of allocated block of memory */</code>

**Assembly Interface** none

**Description** MEM\_alloc allocates a contiguous block of storage from the memory segment identified by segid and returns the address of this block.

The segid parameter identifies the memory segment from which memory is to be allocated. This identifier can be an integer or a memory segment name defined in the Configuration Tool. The files created by the Configuration Tool define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

MEM\_alloc does not initialize the allocated memory locations.

If the memory request cannot be satisfied, MEM\_alloc calls SYS\_error with SYS\_EALLOC and returns MEM\_ILLEGAL.

Memory management functions require that the caller obtain a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_alloc cannot be called from an SWI or HWI.
- ☐ align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

**See Also**

MEM\_calloc  
 MEM\_free  
 MEM\_valloc  
 SYS\_error  
 C library stdlib.h

**MEM\_calloc**

*Allocate from a memory segment and set value to 0*

**C Interface**

<b>Syntax</b>	addr = MEM_calloc(segid, size, align)		
<b>Parameters</b>	Int	segid;	/* memory segment identifier */
	Uns	size;	/* block size in MADUs */
	Uns	align;	/* block alignment */
<b>Return Value</b>	Void	*addr;	/* address of allocated block of memory */
<b>Assembly Interface</b>	none		
<b>Description</b>	MEM_calloc is functionally equivalent to calling MEM_valloc with value set to 0.		

MEM\_calloc allocates a contiguous block of storage from the memory segment identified by segid and returns the address of this block.

The segid parameter identifies the memory segment from which memory is to be allocated. This identifier can be an integer or a memory segment name defined in the Configuration Tool. The files created by the Configuration Tool define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM\_calloc calls SYS\_error with SYS\_EALLOC and returns MEM\_ILLEGAL.

Memory management functions require that the caller obtain a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_calloc cannot be called from an SWI or HWI.
- ☐ align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

**See Also**

MEM\_alloc  
MEM\_free  
MEM\_valloc  
SYS\_error  
C library stdlib.h

**MEM\_define**

*Define a new memory segment*

**C Interface**

<b>Syntax</b>	segid = MEM_define(base, length, attrs);		
<b>Parameters</b>	Ptr	base;	/* base address of new segment */
	Uns	length;	/* length (in MADUs) of new segment */
	MEM_Attrs	*attrs;	/* segment attributes */
<b>Return Value</b>	Int	segid;	/* ID of new segment */

**Assembly Interface** none

**Description** MEM\_define defines a new memory segment for use by the DSP/BIOS memory module, MEM.

The new segment contains length MADUs starting at base. A new table entry is allocated to define the segment, and the entry's index into this table is returned as the segid.

The new block should be aligned on a MEM\_HEADERSIZE boundary, and the length should be a multiple of MEM\_HEADERSIZE, otherwise the entire block is not available for allocation.

If attrs is NULL, the new segment is assigned a default set of attributes. Otherwise, the segment's attributes are specified through a structure of type MEM\_Attrs.

**Note:**

No attributes are supported for segments, and the type MEM\_Attrs is defined as a dummy structure.

**Constraints and Calling Context**

- ❑ At least one segment must exist at the time MEM\_define is called.
- ❑ MEM\_define and MEM\_redefine must not be called when a context switch is possible. To guard against a context switch, these functions should only be called in the main function.

**See Also** MEM\_redefine

MEM\_free

Free a block of memory

C Interface

Syntax	status = MEM_free(segid, addr, size);		
Parameters	Int	segid;	/* memory segment identifier */
	Ptr	addr;	/* block address pointer */
	Uns	size;	/* block length in MADUs*/
Return Value	Bool	status;	/* TRUE if successful */
Assembly Interface	none		
Description	MEM_free places the memory block specified by addr and size back into the free pool of the segment specified by segid. This space is then available for further allocation by MEM_alloc. The segid can be an integer or a memory segment name defined in the Configuration Tool		
	Memory management functions require that the caller obtain a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.		
Constraints and Calling Context	<input type="checkbox"/> addr must be a valid pointer returned from a call to MEM_alloc.		
	<input type="checkbox"/> segid and size are those values used in a previous call to MEM_alloc.		
	<input type="checkbox"/> MEM_free cannot be called by HWI or SWI functions.		
See Also	MEM_alloc		
	C library stdlib.h		

**MEM\_redefine***Redefine an existing memory segment***C Interface****Syntax**

MEM\_redefine(segid, base, length);

**Parameters**

Int            segid;     /\* segment to redefine \*/  
 Ptr          base;     /\* base address of new block \*/  
 Uns          length;   /\* length (in MADUs) of new block \*/

**Return Value**

Void

**Assembly Interface**

none

**Reentrant**

no

**Description**

MEM\_redefine redefines an existing memory segment managed by the DSP/BIOS memory module, MEM. All pointers in the old segment memory block are automatically freed, and the new segment block is completely available for allocations.

The new block should be aligned on a MEM\_HEADERSIZE boundary, and the length should be a multiple of MEM\_HEADERSIZE, otherwise the entire block is not available for allocation.

**Constraints and Calling Context**

- ❑ MEM\_define and MEM\_redefine must not be called when a context switch is possible. To guard against a context switch, these functions should only be called in the main function.

**See Also**

MEM\_define



**MEM\_stat***Return the status of a memory segment***C Interface****Syntax**`status = MEM_stat(segid, statbuf);`**Parameters**

Int	segid;	/* memory segment identifier */
MEM_Stat	*statbuf;	/* pointer to stat buffer */

**Return Value**

Bool	status;	/* TRUE if successful */
------	---------	--------------------------

**Assembly Interface**

none

**Description**

MEM\_stat returns the status of the memory segment specified by segid in the status structure pointed to by statbuf.

```
struct MEM_Stat {
    Uns  size;    /* original size of segment */
    Uns  used;    /* number of MADUs used in segment */
    Uns  length;  /* largest free contiguous block length */
}
```

All values are expressed in terms of minimum addressable units (MADUs).

MEM\_stat returns TRUE if segid corresponds to a valid memory segment, and FALSE otherwise. If MEM\_stat returns FALSE, the contents of statbuf are undefined.

Memory management functions require that the caller obtain a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

**Constraints and  
Calling Context**

- ❑ MEM\_stat cannot be called from an SWI or HWI.

**MEM\_valloc***Allocate from a memory segment and set value***C Interface**

<b>Syntax</b>	<code>addr = MEM_valloc(segid, size, align, value);</code>		
<b>Parameters</b>	<code>Int</code>	<code>segid;</code>	<i>/* memory segment identifier */</i>
	<code>Uns</code>	<code>size;</code>	<i>/* block size in MADUs */</i>
	<code>Uns</code>	<code>align;</code>	<i>/* block alignment */</i>
	<code>Char</code>	<code>value;</code>	<i>/* character value */</i>
<b>Return Value</b>	<code>Void</code>	<code>*addr;</code>	<i>/* address of allocated block of memory */</i>

**Assembly Interface** none**Description** MEM\_valloc uses MEM\_alloc to allocate the memory before initializing it to value.

The segid parameter identifies the memory segment from which memory is to be allocated. This identifier can be an integer or a memory segment name defined in the Configuration Tool. The files created by the Configuration Tool define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM\_valloc calls SYS\_error with SYS\_EALLOC and returns MEM\_ILLEGAL.

Memory management functions require that the caller obtain a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_valloc cannot be called from an SWI or HWI.
- ☐ align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

**See Also**

MEM\_alloc  
 MEM\_calloc  
 MEM\_free  
 SYS\_error  
 C library `stdlib.h`

## 2.14 PIP Module

The PIP module is the buffered pipe manager.

### Functions

- ☐ **PIP\_alloc.** Get an empty frame from the pipe.
- ☐ **PIP\_free.** Recycle a frame back to the pipe.
- ☐ **PIP\_get.** Get a full frame from the pipe.
- ☐ **PIP\_getReaderAddr.** Get the value of the readerAddr pointer of the pipe.
- ☐ **PIP\_getReaderNumFrames.** Get the number of pipe frames available for reading.
- ☐ **PIP\_getReaderSize.** Get the number of words of data in a pipe frame.
- ☐ **PIP\_getWriterAddr.** Get the value of the writerAddr pointer of the pipe.
- ☐ **PIP\_getWriterNumFrames.** Get the number of pipe frames available to write to.
- ☐ **PIP\_getWriterSize.** Get the number of words that can be written to a pipe frame.
- ☐ **PIP\_peek.** Get the pipe frame size and address without actually claiming the pipe frame.
- ☐ **PIP\_put.** Put a full frame into the pipe.
- ☐ **PIP\_reset.** Reset all fields of a pipe object to their original values.
- ☐ **PIP\_setWriterSize.** Set the number of valid words written to a pipe frame.

### PIP\_Obj Structure Members

- ☐ **Ptr readerAddr.** Pointer to the address to begin reading from after calling PIP\_get.
- ☐ **Uns readerSize.** Number of words of data in the frame read with PIP\_get.
- ☐ **Uns readerNumFrames.** Number of frames available to be read.
- ☐ **Ptr writerAddr.** Pointer to the address to begin writing to after calling PIP\_alloc.
- ☐ **Uns writerSize.** Number of words available in the frame allocated with PIP\_alloc.
- ☐ **Uns writerNumFrames.** Number of frames available to be written to.

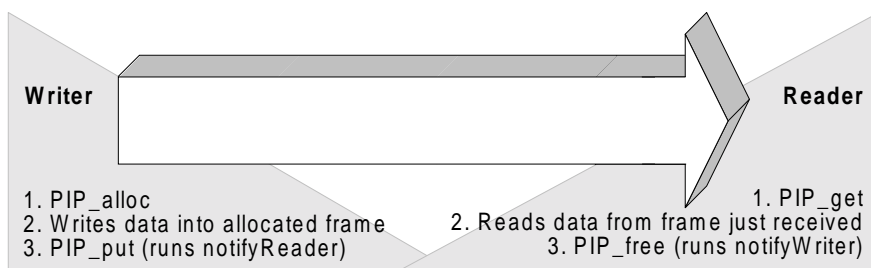
## Description

The PIP module manages data pipes, which 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 device and all kinds of real-time peripheral devices.

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 can put a variable amount of data in each frame up to the length of the frame.

A pipe has two ends, as shown in Figure 2-2. The writer end (also called the producer) is where your program writes frames of data. The reader end (also called the consumer) is where your program reads frames of data.

Figure 2-2. Pipe Schematic



Internally, pipes are implemented as a circular list; frames are reused at the writer end of the pipe after `PIP_free` releases them.

The `notifyReader` and `notifyWriter` functions are called from the context of the code that calls `PIP_put` or `PIP_free`. These functions can be written in C or assembly. To avoid problems with recursion, the `notifyReader` and `notifyWriter` functions normally should not directly call any of the PIP module functions for the same pipe. Instead, they should post a software interrupt that uses the PIP module functions. However, PIP calls may be made from the `notifyReader` and `notifyWriter` functions if the functions have been protected against re-entrancy. The audio example, located on your distribution CD in `c:\ti\examples\target\bios\audio` folder, where `target` matches your board, is a good example of this. (If you installed in a path other than `c:\ti`, substitute your appropriate path.)

**Note:**

When DSP/BIOS starts up, it calls the notifyWriter function internally for each created pipe object to initiate the pipe's I/O.

The code that calls PIP\_free or PIP\_put should preserve any necessary registers.

Often one end of a pipe is controlled by an HWI and the other end is controlled by a SWI function.

HST objects use PIP objects internally for I/O between the host and the target. Your program only needs to act as the reader or the writer when you use an HST object, because the host controls the other end of the pipe.

Pipes can also be used to transfer data within the program between two application threads.

**PIP Manager Properties**

The pipe manager manages objects that allow the efficient transfer of frames of data between a single reader and a single writer. This transfer is often between an HWI and an application software interrupt, but pipes can also be used to transfer data between two application threads.

The following global property can be set for the PIP module on the PIP Manager Properties dialog in the Configuration Tool:

**Object Memory.** The memory segment that contains the PIP objects.

**PIP Object Properties**

A pipe object maintains a single contiguous buffer partitioned into a fixed number of fixed length frames. All I/O operations on a pipe deal with one frame at a time; although each frame has a fixed length, the application can put a variable amount of data in each frame (up to the length of the frame).

The following properties can be set for a pipe object on the PIP Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this PIP object.
- ☐ **bufseg.** The memory segment that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- ☐ **bufalign.** The alignment (in words) of the buffer allocated within the specified memory segment.
- ☐ **framesize.** The length of each frame (in words)

- ❑ **numframes.** The number of frames
- ❑ **monitor.** The end of the pipe to be monitored by a hidden STS object. Can be set to reader, writer, or nothing. In the Statistics View analysis tool, your choice determines whether the STS display for this pipe shows a count of the number of frames handled at the reader or writer end of the pipe.
- ❑ **notifyWriter.** The function to execute when a frame of free space is available. This function should notify (for example, by calling SWI\_andn) the object that writes to this pipe that an empty frame is available.  
The notifyWriter function is performed as part of the thread that called PIP\_free or PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any of the PIP module functions for the same pipe.
- ❑ **nwarg0, nwarg1.** Two Arg type arguments for the notifyWriter function.
- ❑ **notifyReader.** The function to execute when a frame of data is available. This function should notify (for example, by calling SWI\_andn) the object that reads from this pipe that a full frame is ready to be processed.  
The notifyReader function is performed as part of the thread that called PIP\_put or PIP\_get. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.
- ❑ **nrarg0, nrarg1.** Two Arg type arguments for the notifyReader function.

## PIP - Code Composer Studio Interface

To enable PIP accumulators, choose DSP/BIOS→RTA Control Panel and put a check in the appropriate box. Then choose DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a PIP object, you see a count of the number of frames read from or written to the pipe.

**PIP\_alloc**

*Allocate an empty frame from a pipe*

**C Interface**

<b>Syntax</b>	PIP_alloc(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	PIP_alloc
<b>Preconditions</b>	ar2 = address of the pipe object the pipe must contain empty frames before calling PIP_alloc
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm

**Assembly Interface**



<b>Syntax</b>	PIP_alloc
<b>Preconditions</b>	xar0 = the address of the pipe object; the pipe must contain empty frames before calling PIP_alloc
<b>Postconditions</b>	none
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, t1, ac0, ac1, ac2, ac3, rptc, trn1, brc1, brs1, csr, rsa0, rsa1, rea0, rea1 and any registers modified by the notifyWriter function

**Reentrant** no

**Description** PIP\_alloc allocates an empty frame from the pipe you specify. You can write to this frame and then use PIP\_put to put the frame into the pipe.

If empty frames are available after PIP\_alloc allocates a frame, PIP\_alloc runs the function specified by the notifyWriter property of the PIP object. This function should notify (for example, by calling SWI\_andn) the object that writes to this pipe that an empty frame is available. The notifyWriter function is performed as part of the thread that calls PIP\_free or

PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any PIP module functions for the same pipe.

## Constraints and Calling Context

- ❑ Before calling PIP\_alloc, a function should check the writerNumFrames member of the PIP\_Obj structure by calling PIP\_getWriterNumFrames to make sure it is greater than 0 (that is, at least one empty frame is available).
- ❑ PIP\_alloc can only be called one time before calling PIP\_put. You cannot operate on two frames from the same pipe simultaneously.

## Example

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

    in = HST_getpipe(input);
    out = HST_getpipe(output);

    if (PIP_getReaderNumFrames(in) == 0 ||
        PIP_getWriterNumFrames(out) == 0) {
        error;
    }

    /* get input data and allocate output frame */
    PIP_get(in);
    PIP_alloc(out);

    /* copy input data to output frame */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);
    size = PIP_getReaderSize(in);
    PIP_setWriterSize(out, size);
    for (; size > 0; size--) {
        *dst++ = *src++;
    }

    /* output copied data and free input frame */
    PIP_put(out);
    PIP_free(in);
}
```

The example for HST\_getpipe, page 2–79, also uses a pipe with host channel objects.

## See Also

PIP\_free  
 PIP\_get  
 PIP\_put  
 HST\_getpipe



PIP\_free

Recycle a frame that has been read to a pipe

C Interface

Syntax	PIP_free(pipe);
Parameters	PIP_Handle pipe; /* pipe object handle */
Return Value	Void

Assembly Interface



Syntax	PIP_free
Preconditions	ar2 = address of the pipe object
Postconditions	none
Modifies	ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, xsm, and any registers modified by the notifyWriter function

Assembly Interface



Syntax	PIP_free
Preconditions	xar0 = address of the pipe object
Postconditions	none
Modifies	xar0, xar1, xar2, xar3, xar4, t0, t1, ac0, ac1, ac2, ac3, rptc, trn1, brc1, brs1, csr, rsa0, rsa1, rea0, rea1, and any registers modified by the notifyWriter function

Reentrant no

Description PIP\_free releases a frame after you have read the frame with PIP\_get. The frame is recycled so that PIP\_alloc can reuse it.

After PIP\_free releases the frame, it runs the function specified by the notifyWriter property of the PIP object. This function should notify (for example, by calling SWI\_andn) the object that writes to this pipe that an empty frame is available. The notifyWriter function is performed as part

of the thread that called PIP\_free or PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any of the PIP module functions for the same pipe.

**Constraints and  
Calling Context**

- ❑ When called within an HWI ISR, the code sequence calling PIP\_free must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**Example**

See the example for PIP\_alloc, page 2–152. The example for HST\_getpipe, page 2–79, also uses a pipe with host channel objects.

**See Also**

PIP\_alloc  
PIP\_get  
PIP\_put  
HST\_getpipe

**PIP\_get**

*Get a full frame from the pipe*

**C Interface**

<b>Syntax</b>	PIP_get(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	PIP_get
<b>Preconditions</b>	ar2 = address of the pipe object the pipe must contain full frames before calling PIP_get
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm

**Assembly Interface**



<b>Syntax</b>	PIP_get
<b>Preconditions</b>	xar0 = address of the pipe object; the pipe must contain full frames before calling PIP_get
<b>Postconditions</b>	none
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, t1, ac0, ac1, ac2, ac3, rptc, trn1, brc1, brs1, csr, rsa0, rsa1, rea0, rea1 and any registers modified by the notifyReader function

**Reentrant** no

**Description** PIP\_get gets a frame from the pipe after some other function puts the frame into the pipe with PIP\_put.

If full frames are available after PIP\_get gets a frame, PIP\_get runs the function specified by the notifyReader property of the PIP object. This function should notify (for example, by calling SWI\_andn) the object that

reads from this pipe that a full frame is available. The notifyReader function is performed as part of the thread that calls PIP\_get or PIP\_put. To avoid problems with recursion, the notifyReader function should not directly call any PIP module functions for the same pipe.

**Constraints and  
Calling Context**

- ❑ Before calling PIP\_get, a function should check the readerNumFrames member of the PIP\_Obj structure by calling PIP\_getReaderNumFrames to make sure it is greater than 0 (that is, at least one full frame is available).
- ❑ PIP\_get can only be called one time before calling PIP\_free. You cannot operate on two frames from the same pipe simultaneously.

**Example**

See the example for PIP\_alloc, page 2–152. The example for HST\_getpipe, page 2–79, also uses a pipe with host channel objects.

**See Also**

PIP\_alloc  
PIP\_free  
PIP\_put  
HST\_getpipe

**PIP\_getReaderAddr***Get the value of the readerAddr pointer of the pipe***C Interface**

<b>Syntax</b>	<code>readerAddr = PIP_getReaderAddr(pipe);</code>
<b>Parameters</b>	<code>PIP_Handle pipe;</code> /* pipe object handle */
<b>Return Value</b>	<code>Ptr readerAddr</code>

**Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	none
<b>Postconditions</b>	none
<b>Modifies</b>	none

<b>Reentrant</b>	yes
------------------	-----

**Description**

PIP\_getReaderAddr is a C function that returns the value of the readerAddr pointer of a pipe object. The readerAddr pointer is normally used following a call to PIP\_get, as the address to begin reading from.

**Example**

```
Void audio(PIP_Obj *in, PIP_Obj *out)
{
    Uns          *src, *dst;
    Uns          size;

    if (PIP_getReaderNumFrames(in) == 0 ||
        PIP_getWriterNumFrames(out) == 0) {
        error;
    }
    PIP_get(in);          /* get input data */
    PIP_alloc(out);       /* allocate output buffer */

    /* copy input data to output buffer */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);
    size = PIP_getReaderSize(in);
    PIP_setWriterSize(out, size);
    for (; size > 0; size--) {
        *dst++ = *src++;
    }

    /* output copied data and free input buffer */
    PIP_put(out);
    PIP_free(in);
}
```

PIP\_getReaderNumFrames

Get the number of pipe frames available for reading

C Interface

Syntax	num = PIP_getReaderNumFrames(pipe);
Parameters	PIP_Handle pipe; /* pip object handle */
Return Value	Uns num; /* number of filled frames to be read */

Assembly Interface

Syntax	none
Preconditions	none
Postconditions	none
Modifies	none

Reentrant                   yes

Description               PIP\_getReaderNumFrames is a C function that returns the value of the readerNumFrames element of a pipe object.

Before a function attempts to read from a pipe it should call PIP\_getReaderNumFrames to ensure at least one full frame is available.

Example                    See the example for PIP\_getReaderAddr, page 2–158.

PIP\_getReaderSize

Get the number of words of data in a pipe frame

C Interface

Syntax	num = PIP_getReaderSize(pipe);
Parameters	PIP_Handle pipe; /* pipe object handle*/
Return Value	Uns num; /* number of words to be read from filled frame */

Assembly Interface

Syntax	none
Preconditions	none
Postconditions	none
Modifies	none
Reentrant	yes

**Description** PIP\_getReaderSize is a C function that returns the value of the readerSize element of a pipe object.

As a function reads from a pipe it should use PIP\_getReaderSize to determine the number of valid words of data in the pipe frame.

**Example** See the example for PIP\_getReaderAddr, page 2–158.

**PIP\_getWriterAddr***Get the value of the writerAddr pointer of the pipe***C Interface**

<b>Syntax</b>	<code>writerAddr = PIP_getWriterAddr(pipe);</code>
<b>Parameters</b>	<code>PIP_Handle pipe;</code> <i>/* pipe object handle */</i>
<b>Return Value</b>	<code>Ptr writerAddr;</code>

**Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	none
<b>Postconditions</b>	none
<b>Modifies</b>	none

**Reentrant**                      yes

**Description**                      PIP\_getWriterAddr is a C function that returns the value of the writerAddr pointer of a pipe object.

The writerAddr pointer is normally used following a call to PIP\_alloc, as the address to begin writing to.

**Example**                              See the example for PIP\_getReaderAddr, page 2–158.



**PIP\_getWriterNumFrames**

*Get number of pipe frames available to be written to*

**C Interface**

<b>Syntax</b>	num = PIP_getWriterNumFrames(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle*/
<b>Return Value</b>	Uns num; /* number of empty frames to be written */

**Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	none
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	yes

**Description** PIP\_getWriterNumFrames is a C function that returns the value of the writerNumFrames element of a pipe object.

Before a function attempts to write to a pipe, it should call PIP\_getWriterNumFrames to ensure at least one empty frame is available.

**Example** See the example for PIP\_getReaderAddr, page 2–158.

## **PIP\_getWriterSize** *Get the number of words that can be written to a pipe frame*

### **C Interface**

<b>Syntax</b>	<code>num = PIP_getWriterSize(pipe);</code>
<b>Parameters</b>	<code>PIP_Handle pipe;</code> <i>/* pipe object handle*/</i>
<b>Return Value</b>	<code>Uns            num;</code> <i>/* number of words to be written in empty frame */</i>

### **Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	none
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	yes
<b>Description</b>	<p>PIP_getWriterSize is a C function that returns the value of the writerSize element of a pipe object.</p> <p>As a function writes to a pipe, it can use PIP_getWriterSize to determine the maximum number words that can be written to a pipe frame.</p>

**Example**

```

if (PIP_getWriterNumFrames(rxPipe) > 0) {
    PIP_alloc(rxPipe);
    DSS_rxPtr = PIP_getWriterAddr(rxPipe);
    DSS_rxCnt = PIP_getWriterSize(rxPipe);
}

```

<b>PIP_peek</b>	<i>Get the pipe frame size and address without actually claiming the pipe frame</i>
<b>C Interface</b>	
<b>Syntax</b>	framesize = PIP_peek(pipe, addr, rw);
<b>Parameters</b>	<div>PIP_Handle pipe; /* pipe object handle */</div> <div>Ptr *addr; /* the address of the variable that keeps the frame address */</div> <div>Uns rw; /* the flag that indicates the reader or writer side */</div>
<b>Return Value</b>	Int framesize; /* the frame size */
<b>Assembly Interface</b>	none
<b>Description</b>	<p>PIP_peek can be used before calling PIP_alloc or PIP_get to get the pipe frame size and address without actually claiming the pipe frame.</p> <p>The pipe parameter is the pipe object handle, the addr parameter is the address of the variable that keeps the retrieved frame address, and the rw parameter is the flag that indicates what side of the pipe PIP_peek is to operate on. If rw is PIP_READER, then PIP_peek operates on the reader side of the pipe. If rw is PIP_WRITER, then PIP_peek operates on the writer side of the pipe.</p> <p>PIP_getReaderNumFrames or PIP_getWriterNumFrames can be called to ensure that a frame exists before calling PIP_peek, although PIP_peek returns -1 if no pipe frame exists.</p> <p>PIP_peek returns the frame size, or -1 if no pipe frames are available. If the return value of PIP_peek in frame size is not -1, then *addr is the location of the frame address.</p>
<b>See Also</b>	<div>PIP_alloc</div> <div>PIP_free</div> <div>PIP_get</div> <div>PIP_put</div> <div>PIP_reset</div>

**PIP\_put***Put a full frame into the pipe***C Interface**

<b>Syntax</b>	PIP_put(pipe);
<b>Parameters</b>	PIP_Handle pipe;     /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PIP_put
<b>Preconditions</b>	ar2 = address of the pipe object
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, xsm, and any registers modified by the notifyReader function

**Assembly Interface**

<b>Syntax</b>	PIP_put
<b>Preconditions</b>	xar0 = address of the pipe object
<b>Postconditions</b>	none
<b>Modifies</b>	xar0, xar1, xar2, xar3, xar4, t0, t1, ac0, ac1, ac2, ac3, rptc, trn1, brc1, brs1, csr, rsa0, rsa1, rea0, rea1, and any registers modified by the notifyReader function

<b>Reentrant</b>	no
------------------	----

<b>Description</b>	PIP_put puts a frame into a pipe after you have allocated the frame with PIP_alloc and written data to the frame. The reader can then use PIP_get to get a frame from the pipe.
--------------------	---

After PIP\_put puts the frame into the pipe, it runs the function specified by the notifyReader property of the PIP object. This function should notify (for example, by calling SWI\_andnHook) the object that reads from this

pipe that a full frame is ready to be processed. The notifyReader function is performed as part of the thread that called PIP\_get or PIP\_put. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.

**Constraints and  
Calling Context**

- ❑ When called within an HWI ISR, the code sequence calling PIP\_put must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**Example**

See the example for PIP\_alloc, page 2–152. The example for HST\_getpipe, page 2–79, also uses a pipe with host channel objects.

**See Also**

PIP\_alloc  
PIP\_free  
PIP\_get  
HST\_getpipe

**PIP\_reset***Reset all fields of a pipe object to their original values***C Interface**

<b>Syntax</b>	PIP_reset(pipe);
<b>Parameters</b>	PIP_Handle pipe;     /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

PIP\_reset resets all fields of a pipe object to their original values.

The pipe parameter specifies the address of the pipe object that is to be reset.

**Constraints and  
Calling Context**

- ❑ PIP\_reset should not be called between the PIP\_alloc call and the PIP\_put call or between the PIP\_get call and the PIP\_free call.
- ❑ PIP\_reset should be called when interrupts are disabled to avoid the race condition.

**See Also**

PIP\_alloc  
PIP\_free  
PIP\_get  
PIP\_peek  
PIP\_put

**PIP\_setWriterSize**

*Set the number of valid words written to a pipe frame*

**C Interface**

<b>Syntax</b>	PIP_setWriterSize(pipe, size);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */ Uns size; /* size to be set */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	none
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	no

**Description** PIP\_setWriterSize is a C function that sets the value of the writerSize element of a pipe object.

As a function writes to a pipe, it can use PIP\_setWriterSize to indicate the number of valid words being written to a pipe frame.

**Example** See the example for PIP\_getReaderAddr, page 2–158.

## 2.15 PRD Module

The PRD module is the periodic function manager.

### Functions

- ☐ `PRD_getticks`. Get the current tick count.
- ☐ `PRD_start`. Arm a periodic function for one-time execution.
- ☐ `PRD_stop`. Stop a periodic function from continuous execution.
- ☐ `PRD_tick`. Advance tick counter, dispatch periodic functions.

### Description

While some applications can schedule functions based on a real-time clock, many applications need to schedule functions based on I/O availability or some other programmatic event.

The PRD module allows you to create PRD objects that schedule periodic execution of program functions. The period can be driven by the CLK module or by calls to `PRD_tick` whenever a specific event occurs. There can be several PRD objects, but all are driven by the same period counter. Each PRD object can execute its functions at different intervals based on the period counter.

- ☐ **To schedule functions based on a real-time clock.** Set the clock interrupt rate you want to use in the Clock Manager property sheet. Put a checkmark in the Use On-chip Clock (CLK) box for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object.
- ☐ **To schedule functions based on I/O availability or some other event.** Remove the checkmark from the Use On-chip Clock (CLK) property field for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object. Your program should call `PRD_tick` to increment the tick counter.

The function executed by a PRD object is statically defined in the Configuration Tool. PRD functions are called from the context of the function run by the `PRD_swi` SWI object. PRD functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

The PRD module uses an SWI object (called `PRD_swi` by default) which itself is triggered on a periodic basis to manage execution of period objects. Normally, this SWI object should have the highest software interrupt priority to allow this software interrupt to be performed once per tick. This software interrupt is automatically created (or deleted) by the Configuration Tool if one or more (or no) PRD objects exist.



See the *Code Composer Studio* online tutorial for an example that demonstrates the interaction between the PRD module and the SWI module.

When the PRD\_swi object runs its function, the following actions occur:

```
for ("Loop through period objects") {  
    if ("time for a periodic function")  
        "run that periodic function";  
}
```

## PRD Manager Properties

The DSP/BIOS Periodic Function Manager allows the creation of an arbitrary number of objects that encapsulate a function, two arguments, and a period specifying the time between successive invocations of the function. The period is expressed in ticks, where a tick is defined as a single invocation of the PRD\_tick operation. The time between successive invocations of PRD\_tick defines the period represented by a tick.

The following global properties can be set for the PRD module on the PRD Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment containing the PRD objects.
- ☐ **Use CLK Manager to drive PRD.** If this field is checked, the on-device timer hardware (managed by CLK) is used to advance the tick count; otherwise, the application must invoke PRD\_tick on a periodic basis.
- ☐ **Microseconds/Tick.** The number of microseconds between ticks. If the Use CLK Manager to drive PRD field above is checked, this field is automatically set by the CLK module; otherwise, you must explicitly set this field.

## PRD Object Properties

The following properties can be set for each PRD object on the PRD Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this PRD object.
- ☐ **period (ticks).** The function executes after period ticks have elapsed.
- ☐ **mode.** If continuous is selected the function executes every period ticks; otherwise it executes just once after each call to PRD\_tick.
- ☐ **function.** The function to be executed
- ☐ **arg0, arg1.** Two Arg type arguments for the user-specified function above.
- ☐ **period (ms).** The number of milliseconds represented by the period specified above. This is an informational field only.

**PRD - Code Composer  
Studio Interface**

To enable PRD logging, choose DSP/BIOS→RTA Control Panel and put a check in the appropriate box. You see indicators for PRD ticks in the PRD ticks row of the Execution Graph, which you can open by choosing DSP/BIOS→Execution Graph. In addition, you see a graph of activity, including PRD function execution.

You can also enable PRD accumulators in the RTA Control Panel. Then you can choose DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a PRD object, you see statistics about the number of ticks elapsed from the time the PRD object is ready to run until it finishes execution. It is important to note, however, if your system is not meeting its timing constraints, the Max value displayed by the Statistics View results in a value that reflects the accumulation of missed deadlines for the PRD object. If Max value becomes greater than the PRD object's period, you can divide Max value by the period to determine how many real-time deadlines your PRD object has missed. While most statistical information can be cleared by right-clicking on the Statistics View and selecting Clear from the pull-down menu, once a periodic function has missed a real-time deadline, the max value will return to its high point as soon as it is recomputed. This is because the information stored about the PRD object used to compute Max value still reflects the fact that the PRD object has missed deadlines.

**PRD\_getticks**

*Get the current tick count*

**C Interface**

<b>Syntax</b>	num = PRD_getticks();
<b>Parameters</b>	Void
<b>Return Value</b>	LgUns          num          /* current tick counter */

**Assembly Interface**



<b>Syntax</b>	PRD_getticks
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE
<b>Postconditions</b>	ah = upper 16 bits of the 32-bit tick counter al = lower 16 bits of the 32-bit tick counter
<b>Modifies</b>	ag, ah, al, c

**Assembly Interface**



<b>Syntax</b>	PRD_getticks
<b>Preconditions</b>	none
<b>Postconditions</b>	ac0h = upper 16 bits of the 32-bit tick counter
<b>Modifies</b>	ac0
<b>Reentrant</b>	yes
<b>Description</b>	PRD_getticks returns the current period tick count as a 32-bit value.

If the periodic functions are being driven by the on-device timer, the tick value is the number of low resolution clock ticks that have occurred since the program started running. When the number of ticks reaches the maximum value that can be stored in 32 bits, the value wraps back to 0. See the CLK Module, page 2–26, for more details.

If the periodic functions are being driven programmatically, the tick value is the number of times PRD\_tick has been called.

**Example**

```
/* ===== showTicks ===== */  
Void showTicks  
{  
    LOG_printf(&trace, "ticks = %d", PRD_getticks());  
}
```

**See Also**

PRD\_start  
PRD\_tick  
CLK\_gettime  
CLK\_gettime  
STS\_delta

**PRD\_start**

*Arm a periodic function for one-shot execution*

**C Interface**

<b>Syntax</b>	PRD_start(prd);
<b>Parameters</b>	PRD_Handle prd;      /* prd object handle*/
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	PRD_start
<b>Preconditions</b>	ar2 = address of the PRD object
<b>Postconditions</b>	none
<b>Modifies</b>	c

**Assembly Interface**



<b>Syntax</b>	PRD_start
<b>Preconditions</b>	xar0 = address of the PRD object
<b>Postconditions</b>	none
<b>Modifies</b>	t0
<b>Reentrant</b>	no

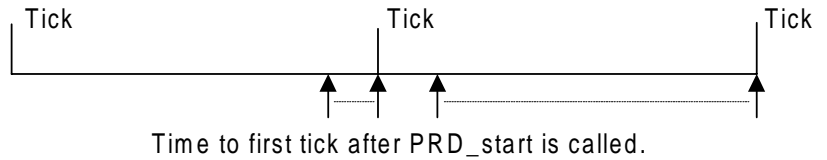
**Description** PRD\_start starts a period object that has its mode property set to one-shot in the Configuration Tool.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified number of ticks have occurred after a call to PRD\_start.

For example, you might have a function that should be executed a certain number of periodic ticks after some condition is met.

When you use PRD\_start to start a period object, the exact time the function runs can vary by nearly one tick cycle. As Figure 2-3 shows, PRD ticks occur at a fixed rate and the call to PRD\_start can occur at any point between ticks

Figure 2-3. PRD Tick Cycles



Due to implementation details, if a PRD function calls PRD\_start for a PRD object that is lower in the list of PRD objects, the function sometimes runs a full tick cycle early.

### Example

```
/* ===== startPRD ===== */
Void startPrd(Int periodID)
{
    if ("condition met") {
        PRD_start(&periodID);
    }
}
```

### See Also

PRD\_tick  
PRD\_getticks

**PRD\_stop**

*Stop a period object to prevent its function execution*

**C Interface**

<b>Syntax</b>	PRD_stop(prd);
<b>Parameters</b>	PRD_Handle prd;      /* prd object handle*/
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	PRD_stop
<b>Preconditions</b>	ar2 = address of the PRD object
<b>Postconditions</b>	none
<b>Modifies</b>	c

**Assembly Interface**



<b>Syntax</b>	PRD_stop
<b>Preconditions</b>	xar0 = address of the PRD object
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	no

<b>Description</b>	<p>PRD_stop stops a period object to prevent its function execution. In most cases, PRD_stop is used to stop a period object that has its mode property set to one-shot in the Configuration Tool.</p> <p>Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified numbers of ticks have occurred after a call to PRD_start.</p>
--------------------	---

PRD\_stop is the way to stop those one-shot PRD objects once started and before their period counters have run out.

**Example**

```
PRD_stop(&prd);
```

**See Also**

PRD\_getticks

PRD\_start

PRD\_tick



PRD\_tick

Advance tick counter, enable periodic functions

C Interface

Syntax	PRD_tick();
Parameters	Void
Return Value	Void

Assembly Interface



Syntax	PRD_tick
Preconditions	intm = 1 cpl = 0 dp = GBL_A_SYSPAGE
Postconditions	dp = GBL_A_SYSPAGE
Modifies	ag, ah, al, bg, bh, bl, c, tc

Assembly Interface



Syntax	PRD_tick
Preconditions	intm = 1
Postconditions	none
Modifies	xar0, xar1, xar2, xar3, xar4, t0, t1, tc1, tc2, ac0

Reentrant                      no

**Description**                      PRD\_tick advances the period counter by one tick. Unless you are driving PRD functions using the on-device clock, PRD objects execute their functions at intervals based on this counter.

For example, a hardware ISR could perform PRD\_tick to notify a periodic function when data is available for processing.

**Constraints and  
Calling Context**

- ❑ All the registers that are modified by this API should be saved and restored, before and after the API is invoked, respectively.
- ❑ When called within an HWI ISR, the code sequence calling PRD\_tick must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.
- ❑ Interrupts need to be disabled before calling PRD\_tick.

**See Also**

PRD\_start  
PRD\_getticks

## 2.16 QUE Module

The QUE module is the atomic queue manager.

### Functions

- ❑ `QUE_create`. Create an empty queue.
- ❑ `QUE_delete`. Delete an empty queue.
- ❑ `QUE_dequeue`. Remove from front of queue (non-atomically).
- ❑ `QUE_empty`. Test for an empty queue.
- ❑ `QUE_enqueue`. Insert at end of queue (non-atomically).
- ❑ `QUE_get`. Remove element from front of queue (atomically).
- ❑ `QUE_head`. Return element at front of queue.
- ❑ `QUE_insert`. Insert in middle of queue (non-atomically).
- ❑ `QUE_new`. Set a queue to be empty.
- ❑ `QUE_next`. Return next element in queue (non-atomically).
- ❑ `QUE_prev`. Return previous element in queue (non-atomically).
- ❑ `QUE_put`. Put element at end of queue (atomically).
- ❑ `QUE_remove`. Remove from middle of queue (non-atomically).

### Constants, Types, and Structures

```
typedef struct QUE_Obj *QUE_Handle; /* queue obj handle */
struct QUE_Attrs{                /* queue attributes */
    Int dummy;                    /* DUMMY */
};

QUE_Attrs QUE_ATTRS = {          /* default attribute values */
    0,
};

typedef QUE_Elem;                /* queue element */
```

### Description

The QUE module makes available a set of functions that manipulate queue objects accessed through handles of type `QUE_Handle`. Each queue contains an ordered sequence of zero or more elements referenced through variables of type `QUE_Elem`, which are generally embedded as the first field within some struct.

For example, the `DEV_Frame` structure which is used by SIO and DEV to enqueue and dequeue I/O buffers is defined as follows:

```

struct DEV_Frame {
    QUE_Elem    link;    /* must be first field! */
    Ptr         addr;
    Uns         size;
}

```

The functions `QUE_put` and `QUE_get` are atomic in that they manipulate the queue with interrupts disabled. These functions can therefore be used to safely share queues between tasks, or between tasks and SWIs or HWIs. All other QUE functions should only be called by tasks, or by tasks and SWIs or HWIs when they are used in conjunction with some mutual exclusion mechanism (for example, `SEM_pend` / `SEM_post`, `TSK_disable` / `TSK_enable`).

Once a queue has been created, use `MEM_alloc` to allocate elements for the queue. You can view examples of this in the program code for `quetest` and `semtest` located on your distribution CD in `c:\ti\examples\target\bios\semtest` folder, where *target* matches your board. (If you installed in a path other than `c:\ti`, substitute your appropriate path.)

## QUE Manager Properties

The following global property can be set for the QUE module on the QUE Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the QUE objects.

## QUE Object Properties

The following property can be set for a QUE object on the QUE Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this QUE object.

**QUE\_create**

Create an empty queue

**C Interface**

<b>Syntax</b>	<code>queue = QUE_create(attrs);</code>
<b>Parameters</b>	<code>QUE_Attrs    *attrs;    /* pointer to queue attributes */</code>
<b>Return Value</b>	<code>QUE_Handle queue;    /* handle for new queue object */</code>
<b>Assembly Interface</b>	none
<b>Description</b>	<p>QUE_create creates a new queue which is initially empty. If successful, QUE_create returns the handle of the new queue. If unsuccessful, QUE_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).</p> <p>If attrs is NULL, the new queue is assigned a default set of attributes. Otherwise, the queue's attributes are specified through a structure of type QUE_Attrs.</p>

---

**Note:**

At present, no attributes are supported for queue objects, and the type QUE\_Attrs is defined as a dummy structure.

---

All default attribute values are contained in the constant QUE\_ATTRS, which can be assigned to a variable of type QUE\_Attrs prior to calling QUE\_create.

You can also create a queue by declaring a variable of type QUE\_Obj and initializing the queue with QUE\_new. You can find an example of this in the semtest code example on your distribution CD in `c:\ti\examples\target\bios\semtest` folder, where *target* matches your board. (If you installed in a path other than `c:\ti`, substitute your appropriate path.)

QUE\_create calls MEM\_alloc to dynamically create the object's data structure. MEM\_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2–136.

**Constraints and  
Calling Context**

- ❑ QUE\_create cannot be called from an SWI or HWI.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MEM\_alloc  
QUE\_empty  
QUE\_delete  
SYS\_error

## QUE\_delete

*Delete an empty queue*

### C Interface

<b>Syntax</b>	QUE_delete(queue);
<b>Parameters</b>	QUE_Handle queue;    /* queue handle */
<b>Return Value</b>	Void

### Assembly Interface

none

### Description

QUE\_delete uses MEM\_free to free the queue object referenced by queue.

QUE\_delete calls MEM\_free to delete the QUE object. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

### Constraints and Calling Context

- ❑ queue must be empty.
- ❑ QUE\_delete cannot be called from an SWI or HWI.
- ❑ No check is performed to prevent QUE\_delete from being used on a statically-created object. If a program attempts to delete a queue object that was created using the Configuration Tool, SYS\_error is called.

### See Also

QUE\_create  
QUE\_empty

QUE_dequeue	<i>Remove from front of queue (non-atomically)</i>
<b>C Interface</b>	
<b>Syntax</b>	elem = QUE_dequeue(queue);
<b>Parameters</b>	QUE_Handle queue; /* queue object handle */
<b>Return Value</b>	Ptr elem; /* pointer to former first element */
<b>Assembly Interface</b>	none
<b>Description</b>	QUE_dequeue removes the element from the front of queue and returns elem.
	<hr/> <p><b>Note:</b></p> <p>QUE_get must be used instead of QUE_dequeue if queue is shared by multiple tasks, or tasks and SWIs or HWIs (unless another mutual exclusion mechanism is used). QUE_get runs atomically and will never be interrupted; QUE_dequeue performs the same action but runs non-atomically. While QUE_dequeue is somewhat faster than QUE_get, you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.</p> <hr/>
<b>See Also</b>	QUE_get



**QUE\_empty**

*Test for an empty queue*

**C Interface**

<b>Syntax</b>	<code>empty = QUE_empty(queue);</code>
<b>Parameters</b>	<code>QUE_Handle queue;</code> <i>/* queue object handle */</i>
<b>Return Value</b>	<code>Bool</code> <code>empty;</code> <i>/* TRUE if queue is empty */</i>
<b>Assembly Interface</b>	<code>none</code>
<b>Description</b>	<code>QUE_empty</code> returns <code>TRUE</code> if there are no elements in queue, and <code>FALSE</code> otherwise.
<b>See Also</b>	<code>QUE_get</code>

**QUE\_enqueue**
*Insert at end of queue (non-atomically)*

**C Interface**

<b>Syntax</b>	QUE_enqueue(queue, elem);
<b>Parameters</b>	QUE_Handle queue;   /* queue object handle */ Ptr           elem;   /* pointer to queue elem */
<b>Return Value</b>	Void

**Assembly Interface**       none

**Description**           QUE\_enqueue inserts elem at the end of queue.

---

**Note:**

QUE\_put must be used instead of QUE\_enqueue if queue is shared by multiple tasks, or tasks and SWIs or HWIs (unless another mutual exclusion mechanism is used). QUE\_put runs atomically and will never be interrupted; QUE\_enqueue performs the same action but runs non-atomically. While QUE\_enqueue is somewhat faster than QUE\_put, you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.

---

**See Also**               QUE\_put

## QUE\_get

*Get element from front of queue (atomically)*

### C Interface

<b>Syntax</b>	<code>elem = QUE_get(queue);</code>
<b>Parameters</b>	<code>QUE_Handle queue;</code> <i>/* queue object handle */</i>
<b>Return Value</b>	<code>Void</code> <code>*elem;</code> <i>/* pointer to former first element */</i>

### Assembly Interface

none

### Description

QUE\_get removes the element from the front of queue and returns elem.

Since QUE\_get manipulates queue with interrupts disabled, queue can be shared by multiple tasks, or by tasks and SWIs or HWIs.

Calling QUE\_get with an empty queue returns the queue itself. This provides a means for using a single atomic action to check if a queue is empty, and to remove and return the first element if it is not empty:

```
if ((QUE_Handle)(elem = QUE_get(q)) != q)
    ` process elem `
```

### See Also

QUE\_create  
 QUE\_empty  
 QUE\_put

**QUE\_head***Return element at front of queue***C Interface****Syntax**`elem = QUE_head(queue);`**Parameters**`QUE_Handle queue;   /* queue object handle */`**Return Value**`QUE_Elem   *elem;   /* pointer to first element */`**Assembly Interface**

none

**Description**

QUE\_head returns a pointer to the element at the front of queue. The element is not removed from the queue.

Calling QUE\_head with an empty queue returns the queue itself.

**See Also**`QUE_create``QUE_empty``QUE_put`

<b>QUE_insert</b>	<i>Insert in middle of queue (non-atomically)</i>
<b>C Interface</b>	
<b>Syntax</b>	QUE_insert(qelem, elem);
<b>Parameters</b>	<div>Ptr            qelem;    /* element already in queue */</div> <div>Ptr            elem;     /* element to be inserted in queue */</div>
<b>Return Value</b>	Void
<b>Assembly Interface</b>	none
<b>Description</b>	<div>QUE_insert inserts elem in the queue in front of qelem.</div> <div><b>Note:</b> If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_insert should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/ SEM_post, TSK_disable / TSK_enable).</div>
<b>See Also</b>	<div>QUE_head</div> <div>QUE_next</div> <div>QUE_prev</div> <div>QUE_remove</div>

**QUE\_new***Set a queue to be empty***C Interface**

<b>Syntax</b>	QUE_new(queue);
<b>Parameters</b>	QUE_Handle queue;    /* pointer to queue object */
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

QUE\_new adjusts a queue object to make the queue empty. This operation is not atomic. A typical use of QUE\_new is to initialize a queue object that has been statically declared instead of being created with QUE\_create. Note that if the queue is not empty, the element(s) in the queue are not freed or otherwise handled, but are simply abandoned.

If you created a queue by declaring a variable of type QUE\_Obj, you can initialize the queue with QUE\_new. You can find an example of this in the semtest code example on your distribution CD in `c:\ti\examples\target\bios\semtest` folder, where *target* matches your board. (If you installed in a path other than `c:\ti`, substitute your appropriate path.)

**See Also**

QUE\_create  
QUE\_delete  
QUE\_empty

<b>QUE_next</b>	<i>Return next element in queue (non-atomically)</i>		
<b>C Interface</b>			
<b>Syntax</b>	elem = QUE_next(qelem);		
<b>Parameters</b>	Ptr	qelem;	/* element in queue */
<b>Return Value</b>	Ptr	elem;	/* next element in queue */
<b>Assembly Interface</b>	none		
<b>Description</b>	<p>QUE_next returns elem which points to the element in the queue after qelem.</p> <p>Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE_next to return a pointer to the queue itself. Be careful not to call QUE_remove(elem) in this case.</p> <div><b>Note:</b> If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_next should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/ SEM_post, TSK_disable / TSK_enable).</div>		
<b>See Also</b>	QUE_get QUE_insert QUE_prev QUE_remove		

**QUE\_prev***Return previous element in queue (non-atomically)***C Interface**

<b>Syntax</b>	<code>elem = QUE_prev(qelem);</code>
<b>Parameters</b>	Ptr            qelem;    /* element in queue */
<b>Return Value</b>	Ptr            elem;     /* previous element in queue */

**Assembly Interface**

none

**Description**

QUE\_prev returns elem which points to the element in the queue before qelem.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE\_prev to return a pointer to the queue itself. Be careful not to call QUE\_remove(elem) in this case.

**Note:**

If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE\_prev should be used in conjunction with some mutual exclusion mechanism (for example, SEM\_pend/ SEM\_post, TSK\_disable / TSK\_enable).

**See Also**

QUE\_head  
 QUE\_insert  
 QUE\_next  
 QUE\_remove



**QUE\_put**

*Put element at end of queue (atomically)*

**C Interface**

<b>Syntax</b>	QUE_put(queue, elem);
<b>Parameters</b>	QUE_Handle queue; /* queue object handle */ Void *elem; /* pointer to new queue element */
<b>Return Value</b>	Void
<b>Assembly Interface</b>	none
<b>Description</b>	QUE_put puts elem at the end of queue.  Since QUE_put manipulates queue with interrupts disabled, queue can be shared by multiple tasks, or by tasks and SWIs or HWIs.
<b>See Also</b>	QUE_get QUE_head

## QUE\_remove

*Remove from middle of queue (non-atomically)*

### C Interface

<b>Syntax</b>	QUE_remove(qelem);
<b>Parameters</b>	Ptr            qelem;    /* element in queue */
<b>Return Value</b>	Void

### Assembly Interface

none

### Description

QUE\_remove removes qelem from the queue.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, be careful not to remove the header node. This can happen when qelem is the return value of QUE\_next or QUE\_prev. The following code sample shows how qelem should be verified before calling QUE\_remove.

```
QUE_Elem *qelem;

/* get pointer to first element in the queue */
qelem = QUE_head(queue);

/* scan entire queue for desired element */
while (qelem != queue) {
    if(' qelem is the elem we're looking for ') {
        break;
    }
    qelem = QUE_next(queue);
}

/* make sure qelem is not the queue itself */
if (qelem != queue) {
    QUE_remove(qelem);
}
```

---

#### **Note:**

If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE\_remove should be used in conjunction with some mutual exclusion mechanism (for example, SEM\_pend/ SEM\_post and TSK\_disable / TSK\_enable).

---

**Constraints and  
Calling Context**

QUE\_remove should not be called when qelem is equal to the queue itself.

**See Also**

QUE\_head  
QUE\_insert  
QUE\_next  
QUE\_prev

## 2.17 RTDX Module

The RTDX modules manage the real-time data exchange settings.

### RTDX Data Declaration Macros

- ☐ RTDX\_CreateInputChannel
- ☐ RTDX\_CreateOutputChannel

### Function Macros

- ☐ RTDX\_disableInput
- ☐ RTDX\_disableOutput
- ☐ RTDX\_enableInput
- ☐ RTDX\_enableOutput
- ☐ RTDX\_read
- ☐ RTDX\_readNB
- ☐ RTDX\_sizeofInput
- ☐ RTDX\_write

### Channel Test Macros

- ☐ RTDX\_channelBusy
- ☐ RTDX\_isInputEnabled
- ☐ RTDX\_isOutputEnabled

### Description

The RTDX module provides the data types and functions for:

- ☐ Sending data from the target to the host.
- ☐ Sending data from the host to the target.

Data channels are represented by globally declared structures. A data channel can be used either for input or output, but not both. The contents of an input or output structure are not known to the user. A channel structure contains two states: enabled and disabled. When a channel is enabled, any data written to the channel is sent to the host. Channels are initialized to be disabled.

The RTDX assembly interface, *rtdx.i*, is a macro interface file that can be used to interface to RTDX at the assembly level.

### RTDX Manager Properties

The following settings refer to target configuration parameters:

- ☐ **Enable Real-Time Data Exchange (RTDX).** This box should be checked if you want to link RTDX support into your application.
- ☐ **RTDX Mode.** Select the port configuration mode RTDX should use to establish communication between the host and target. The default is JTAG for most targets. Set this to simulator if you are using a simulator. The HS-RTDX emulation technology is also available. If this property is set incorrectly you will see a message that says "RTDX target application does not match emulation protocol" when you load the program.

- ❑ **RTDX Data Segment (.rtdx\_data).** The memory segment used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this segment.
- ❑ **RTDX Buffer Size (MADUs).** The size of the RTDX target-to-host message buffer, in minimum addressable data units (MADUs). The default size is 1032 to accommodate a 1024-byte block and two control words. HST channels using RTDX are limited by this value.
- ❑ **RTDX Interrupt Mask.** This mask identifies RTDX clients and protects RTDX critical sections. The mask specifies the interrupts to be temporarily disabled inside RTDX critical sections. This also temporarily disables other RTDX clients and prevents another RTDX function call. See the RTDX on-line help for details.

## RTDX Object Properties

The following properties can be set for an RTDX object on the RTDX Object Properties dialog in the Configuration Tool:

- ❑ **comment.** Type a comment to identify this RTDX object.
- ❑ **Channel Mode.** Select output if the RTDX channel handles output from the DSP to the host. Select input if the RTDX channel handles input to the DSP from the host.

## Examples

An Excel example is located in the `c:\ti\examples\hostapps\rtdx` folder. The file is called `rtdx.xls`. (If you installed in a path other than `c:\ti`, substitute your appropriate path.)

The examples are described below.

- ❑ **Ta\_write.asm.** Target to Host transmission example. This example sends 100 consecutive integers starting from 0. In the `rtdx.xls` file, use the `h_read` VB macro to view data on the host.
- ❑ **Ta\_read.asm.** Host to target transmission example. This example reads 100 integers. Use the `h_write` VB macro of the `rtdx.xls` file to send data to the target.
- ❑ **Ta\_readNB.asm.** Host to target transmission example. This example reads 100 integers. Use the `h_write` VB macro of the `rtdx.xls` file to send data to the target. This example demonstrates how to use the non-blocking read, `RTDX_readNB`, function.

---

### Note:

Programs must be linked with C run-time libraries and contain the symbol `_main`.

---

**RTDX\_channelBusy** *Return status indicating whether data channel is busy*

### C Interface

**Syntax** `int RTDX_channelBusy( RTDX_inputChannel *pichan );`

**Parameters** `pichan` */\* Identifier for the input data channel \*/*

**Return Value** `int` */\* Status: 0 = Channel is not busy. \*/  
/\* non-zero = Channel is busy. \*/*

**Assembly Interface** `none`



### Assembly Interface



**Syntax** `RTDX_ChannelBusy .macro pichan`

**Preconditions** `expect the TC bit to be modified by this macro`

**Postconditions** `none`

**Modifies** `TC`

**Reentrant** `yes`

**Description** RTDX\_channelBusy is designed to be used in conjunction with RTDX\_readNB. The return value indicates whether the specified data channel is currently in use or not. If a channel is busy reading, the test/control flag (TC) bit of status register 0 (STO) is set to 1. Otherwise, the TC bit is set to 0.

**Constraints and Calling Context** ☐ RTDX\_channelBusy cannot be called by an HWI function.

**See Also** `RTDX_readNB`

RTDX\_CreateInputChannel

Declare input channel structure

C Interface

Syntax	RTDX_CreateInputChannel( ichan );
Parameters	ichan                      /* Label for the input channel */
Return Value	none
Assembly Interface	none



Assembly Interface



Syntax	RTDX_CreateInputChannel .macro ichan
Preconditions	expect this macro to declare 3 words in the .bss section and initialize to zero
Postconditions	none
Modifies	3 words in the .bss section
Reentrant	no
Description	<p>This macro declares and initializes to 0, the RTDX data channel for input.</p> <p>Data channels must be declared as global objects. A data channel can be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.</p> <p>A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.</p> <p>Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer or its COM interface.</p>
Constraints and Calling Context	<div><input type="checkbox"/> RTDX_CreateInputChannel cannot be called by an HWI function.</div>
See Also	RTDX_CreateOutputChannel

## RTDX\_CreateOutputChannel *Declare output channel structure*

### C Interface

<b>Syntax</b>	RTDX_CreateOutputChannel( ochan );
<b>Parameters</b>	ochan                    /* Label for the output channel */
<b>Return Value</b>	none
<b>Assembly Interface</b>	none



### Assembly Interface



<b>Syntax</b>	RTDX_CreateOutputChannel .macro ochan
<b>Preconditions</b>	expect this macro to declare one word in the .bss section and initialize to zero
<b>Postconditions</b>	none
<b>Modifies</b>	one word in the .bss section
<b>Reentrant</b>	no

<b>Description</b>	<p>This macro declares and initializes the RTDX data channels for output.</p> <p>Data channels must be declared as global objects. A data channel can be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.</p> <p>A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.</p> <p>Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer Studio or its OLE interface.</p>
--------------------	---

<b>Constraints and Calling Context</b>	<ul style="list-style-type: none"> <li><input type="checkbox"/> RTDX_CreateOutputChannel cannot be called by an HWI function.</li> </ul>
--	--

<b>See Also</b>	RTDX_CreateInputChannel
-----------------	-------------------------



RTDX\_disableInput

Disable an input data channel

C Interface

Syntax	void RTDX_disableInput( RTDX_inputChannel *ichan );
Parameters	ichan                      /* Identifier for the input data channel */
Return Value	void
Assembly Interface	none



Assembly Interface



Syntax	RTDX_DisableInput .macro ichan
Preconditions	Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.
Postconditions	none
Modifies	CPL bit and register A
Reentrant	yes
Description	A call to a disable function causes the specified input channel to be disabled.
Constraints and Calling Context	<div><input type="checkbox"/> RTDX_disableInput cannot be called by an HWI function.</div>
See Also	RTDX_disableOutput RTDX_enableInput RTDX_read

## RTDX\_disableOutput *Disable an output data channel*

### C Interface

<b>Syntax</b>	<code>void RTDX_disableOutput( RTDX_outputChannel *ochan );</code>
<b>Parameters</b>	<code>ochan</code> <i>/* Identifier for an output data channel */</i>
<b>Return Value</b>	<code>void</code>
<b>Assembly Interface</b>	<code>none</code>



### Assembly Interface



<b>Syntax</b>	<code>RTDX_DisableOutput .macro ochan</code>
<b>Preconditions</b>	Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.
<b>Postconditions</b>	<code>none</code>
<b>Modifies</b>	CPL bit and register A
<b>Reentrant</b>	<code>yes</code>
<b>Description</b>	A call to a disable function causes the specified data channel to be disabled.
<b>Constraints and Calling Context</b>	<ul style="list-style-type: none"> <li>❑ RTDX_disableOutput cannot be called by an HWI function.</li> </ul>
<b>See Also</b>	RTDX_disableInput RTDX_enableOutput RTDX_read

RTDX\_enableInput

Enable an input data channel

C Interface

Syntax	void RTDX_enableInput( RTDX_inputChannel *ichan );	
Parameters	ochan	/* Identifier for an output data channel */
	ichan	/* Identifier for the input data channel */
Return Value	void	
Assembly Interface	none	



Assembly Interface



Syntax	RTDX_EnableInput .macro ichan
Preconditions	Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.
Postconditions	none
Modifies	CPL bit and register A
Reentrant	yes
Description	A call to an enable function causes the specified data channel to be enabled.
Constraints and Calling Context	<div><input type="checkbox"/> RTDX_enableInput cannot be called by an HWI function.</div>
See Also	RTDX_disableInput RTDX_enableOutput RTDX_read

**RTDX\_enableOutput***Enable an output data channel***C Interface**

<b>Syntax</b>	<code>void RTDX_enableOutput( RTDX_outputChannel *ochan );</code>
<b>Parameters</b>	<code>ochan</code> <i>/* Identifier for an output data channel */</i>
<b>Return Value</b>	<code>void</code>
<b>Assembly Interface</b>	<code>none</code>

**Assembly Interface**

<b>Syntax</b>	<code>RTDX_EnableOutput .macro ochan</code>
<b>Preconditions</b>	Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.
<b>Postconditions</b>	<code>none</code>
<b>Modifies</b>	CPL bit and register A
<b>Reentrant</b>	<code>yes</code>
<b>Description</b>	A call to an enable function causes the specified data channel to be enabled.
<b>Constraints and Calling Context</b>	❑ RTDX_enableOutput cannot be called by an HWI function.
<b>See Also</b>	RTDX_disableOutput RTDX_enableInput RTDX_write

RTDX\_isInputEnabled

Return status of the input data channel

C Interface

Syntax	RTDX_isInputEnabled( ichan );	
Parameter	ichan	/* Identifier for an input channel. */
Return Value	0	/* Not enabled. */
	non-zero	/* Enabled. */

Assembly Interface      none



Assembly Interface



Syntax	RTDX_isInputEnabled .macro ichan
Preconditions	Expect the TC bit to be modified by this macro.
Postconditions	none
Modifies	TC bit
Reentrant	yes
Description	The RTDX_isInputEnabled macro tests to see if an input channel is enabled and sets the test/control flag (TC bit) of status register 0 to 1 if the input channel is enabled. Otherwise, it sets the TC bit to 0.
Constraints and Calling Context	<div><input type="checkbox"/> RTDX_isInputEnabled cannot be called by an HWI function.</div>
See Also	RTDX_isOutputEnabled

RTDX\_isOutputEnabled

Return status of the output data channel

C Interface

Syntax	RTDX_isOutputEnabled(ohan );	
Parameter	ohan	/* Identifier for an output channel. */
Return Value	0	/* Not enabled. */
	non-zero	/* Enabled. */

Assembly Interface

none



Assembly Interface



Syntax	RTDX_isOutputEnabled .macro ochan
Preconditions	Expect the TC bit to be modified by this macro.
Postconditions	none
Modifies	TC bit
Reentrant	yes

**Description** The RTDX\_isOutputEnabled macro tests to see if an output channel is enabled and sets the test/control flag (TC bit) of status register 0 to 1 if the output channel is enabled. Otherwise, it sets the TC bit to 0.

**Constraints and Calling Context** ☐ RTDX\_isOutputEnabled cannot be called by an HWI function.

**See Also** RTDX\_isInputEnabled

RTDX\_read

Read from an input channel

C Interface

Syntax	int RTDX_read( RTDX_inputChannel *ichan, void *buffer, int bsize );	
Parameters	ichan	/* Identifier for the input data channel */
	buffer	/* A pointer to the buffer that receives the data */
	bsize	/* The size of the buffer in address units */
Return Value	> 0	/* The number of address units of data */
		/* actually supplied in buffer. */
	0	/* Failure. Cannot post read request */
		/* because target buffer is full. */
	RTDX_READ_ERROR	/* Failure. Channel currently busy or not enabled. */

Assembly Interface      none



Assembly Interface



Syntax      RTDX\_read .macro ichan, buffer, bsize

Preconditions      Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit, register BL, and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.

Postconditions      The return value of the read is placed in the accumulator (register A).

Modifies      CPL bit, register BL, and register A

Reentrant      yes

Description      RTDX\_read causes a read request to be posted to the specified input data channel. If the channel is enabled, RTDX\_read waits until the data has arrived. On return from the function, the data has been copied into the specified buffer and the number of address units of data actually supplied is returned. The function returns RTDX\_READ\_ERROR immediately if the channel is currently busy reading or is not enabled.

When RTDX\_read is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host

Library to write data to the target buffer. When the data is received, the target application continues execution.

The specified data is to be written to the specified output data channel, provided that channel is enabled. On return from the function, the data has been copied out of the specified user buffer and into the RTDX target buffer. If the channel is not enabled, the write operation is suppressed. If the RTDX target buffer is full, failure is returned.

When RTDX\_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data, but the target application does not wait. Execution of the target application continues immediately. Use RTDX\_channelBusy and RTDX\_sizeofInput to determine when the RTDX Host Library has written data to the target buffer.

### **Constraints and Calling Context**

- ❑ RTDX\_read cannot be called by an HWI function.

### **See Also**

RTDX\_channelBusy  
RTDX\_readNB  
RTDX\_sizeofInput



RTDX\_readNB

Read from input channel without blocking

C Interface

Syntax	int RTDX_readNB( RTDX_inputChannel *ichan, void *buffer, int bsize );	
Parameters	ichan	/* Identifier for the input data channel */
	buffer	/* A pointer to the buffer that receives the data */
	bsize	/* The size of the buffer in address units */
Return Value	RTDX_OK	/* Success.*/
	0 (zero)	/* Failure. The target buffer is full. */
	RTDX_READ_ERROR	/*Channel is currently busy reading. */

Assembly Interface

none



Assembly Interface



Syntax	RTDX_readNB .macro ichan, buffer, bsize
Preconditions	Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit, register BL, and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.
Postconditions	The return value of the read is placed in the accumulator (register A).
Modifies	CPL bit, register BL, and register A
Reentrant	yes

**Description** RTDX\_readNB is a nonblocking form of the function RTDX\_read. RTDX\_readNB issues a read request to be posted to the specified input data channel and immediately returns. If the channel is not enabled or the channel is currently busy reading, the function returns RTDX\_READ\_ERROR. The function returns 0 if it cannot post the read request due to lack of space in the RTDX target buffer.

When the function RTDX\_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data but the target application does not wait. Execution of the target application continues

immediately. Use the `RTDX_channelBusy` and `RTDX_sizeofInput` functions to determine when the RTDX Host Library has written data into the target buffer.

When `RTDX_read` is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host Library to write data into the target buffer. When the data is received, the target application continues execution.

### **Constraints and Calling Context**

- ❑ `RTDX_readNB` cannot be called by an HWI function.

### **See Also**

`RTDX_channelBusy`  
`RTDX_read`  
`RTDX_sizeofInput`

RTDX\_sizeofInput

Return the number of MADUs read from a data channel

C Interface

Syntax	int RTDX_sizeofInput( RTDX_inputChannel *pichan );	
Parameters	pichan	/* Identifier for the input data channel */
Return Value	int	/* Number of sizeof units of data actually */ /* supplied in buffer */

Assembly Interface

none



Assembly Interface



Syntax	RTDX_sizeofInput .macro ichan
Preconditions	Expect register A to be modified by this macro.
Postconditions	The return value of the read is placed in the accumulator (register A).
Modifies	register A

Reentrant

yes

Description

RTDX\_sizeofInput is designed to be used in conjunction with RTDX\_readNB after a read operation has completed. The function returns the number of sizeof units actually read from the specified data channel into the accumulator (register A).

Constraints and Calling Context

❑ RTDX\_sizeofInput cannot be called by an HWI function.

See Also

RTDX\_readNB

**RTDX\_write***Write to an output channel***C Interface**

**Syntax** `int RTDX_write( RTDX_outputChannel *ochan, void *buffer, int bsize );`

**Parameters**

<code>ochan</code>	<code>/* Identifier for the output data channel */</code>
<code>buffer</code>	<code>/* A pointer to the buffer containing the data */</code>
<code>bsize</code>	<code>/* The size of the buffer in address units */</code>

**Return Value** `int` `/* Status: non-zero = Success. 0 = Failure. */`

**Assembly Interface**

none

**Assembly Interface**

**Syntax** `RTDX_write .macro ochan, buffer, bsize`

**Preconditions** Set the CPL bit before this macro calls the RTDX functions. Expect the CPL bit, register BL, and register A to be modified by this macro. Registers ar1, ar6, ar7, and sp can be modified by the function call.

**Postconditions** The return value of the read is placed in the accumulator (register A).

**Modifies** CPL bit, register BL, and register A

**Reentrant** yes

**Description** RTDX\_write causes the specified data to be written to the specified output data channel, provided that channel is enabled. On return from the function, the data has been copied out of the specified user buffer and into the RTDX target buffer. If the channel is not enabled, the write operation is suppressed. If the RTDX target buffer is full, Failure is returned.

**Constraints and Calling Context** ☐ RTDX\_write cannot be called by an HWI function.

**See Also** RTDX\_read

## 2.18 SEM Module

The SEM module is the semaphore manager.

### Functions

- ☐ **SEM\_count.** Get current semaphore count
- ☐ **SEM\_create.** Create a semaphore
- ☐ **SEM\_delete.** Delete a semaphore
- ☐ **SEM\_ipost.** Signal a semaphore (interrupt only)
- ☐ **SEM\_new.** Initialize a semaphore
- ☐ **SEM\_pend.** Wait for a semaphore
- ☐ **SEM\_post.** Signal a semaphore
- ☐ **SEM\_reset.** Reset semaphore

### Constants, Types, and Structures

```
typedef struct SEM_Obj  *SEM_Handle;
                        /* handle for semaphore object */

struct SEM_Attrs { /* semaphore attributes */
    Int    dummy; /* DUMMY */
};

SEM_Attrs SEM_ATTRS = { /* default attribute values */
    0,
};
```

### Description

The SEM module makes available a set of functions that manipulate semaphore objects accessed through handles of type SEM\_Handle. SEM semaphores are counting semaphores that can be used for both task synchronization and mutual exclusion.

SEM\_pend is used to wait for a semaphore. The timeout parameter to SEM\_pend allows the task to wait until a timeout, wait indefinitely, or not wait at all. SEM\_pend's return value is used to indicate if the semaphore was signaled successfully.

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 puts it on the ready queue. If no tasks are waiting, SEM\_post simply increments the semaphore count and returns.

### SEM Manager Properties

The following global property can be set for the SEM module on the SEM Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the SEM objects created with the Configuration Tool.

**SEM Object Properties**

The following properties can be set for a SEM object on the SEM Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this SEM object.
- ☐ **Initial semaphore count.** Set this property to the desired initial semaphore count.

**SEM - Code Composer Studio Interface**

The SEM tab of the Kernel/Object View shows information about semaphore objects.

<b>SEM_count</b>	<i>Get current semaphore count</i>
<b>C Interface</b>	
<b>Syntax</b>	count = SEM_count(sem);
<b>Parameters</b>	SEM_Handle sem;      /* semaphore handle */
<b>Return Value</b>	Int                  count;      /* current semaphore count */
<b>Assembly Interface</b>	none
<b>Description</b>	SEM_count returns the current value of the semaphore specified by sem.

**SEM\_create***Create a semaphore***C Interface**

**Syntax**                    `sem = SEM_create(count, attrs);`

**Parameters**            `Int                    count;        /* initial semaphore count */`  
                              `SEM_Attrs        *attrs;       /* pointer to semaphore attributes */`

**Return Value**           `SEM_Handle sem;       /* handle for new semaphore object */`

**Assembly Interface**

none

**Description**

SEM\_create creates a new semaphore object which is initialized to count. If successful, SEM\_create returns the handle of the new semaphore. If unsuccessful, SEM\_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort).

If attrs is NULL, the new semaphore is assigned a default set of attributes. Otherwise, the semaphore's attributes are specified through a structure of type SEM\_Attrs.

**Note:**

At present, no attributes are supported for semaphore objects, and the type SEM\_Attrs is defined as a dummy structure.

Default attribute values are contained in the constant SEM\_ATTRS, which can be assigned to a variable of type SEM\_Attrs before calling SEM\_create.

SEM\_create calls MEM\_alloc to dynamically create the object's data structure. MEM\_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module.

**Constraints and Calling Context**

- ☐ count must be greater than or equal to 0.
- ☐ SEM\_create cannot be called from an SWI or HWI.
- ☐ You can reduce the size of your application by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MEM\_alloc  
 SEM\_delete  
 SYS\_error



**SEM\_delete**

Delete a semaphore

C Interface

Syntax	SEM_delete(sem);
Parameters	SEM_Handle sem;      /* semaphore object handle */
Return Value	Void
Assembly Interface	none
Description	<p>SEM_delete uses MEM_free to free the semaphore object referenced by sem.</p> <p>SEM_delete calls MEM_free to delete the SEM object. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.</p>
Constraints and Calling Context	<ul style="list-style-type: none"><li>❑ No tasks should be pending on sem when SEM_delete is called.</li><li>❑ SEM_delete cannot be called from an SWI or HWI.</li><li>❑ No check is performed to prevent SEM_delete from being used on a statically-created object. If a program attempts to delete a semaphore object that was created using the Configuration Tool, SYS_error is called.</li></ul>
See Also	SEM_create

**SEM\_ipost***Signal a semaphore (interrupt use only)***C Interface**

<b>Syntax</b>	SEM_ipost(sem);
<b>Parameters</b>	SEM_Handle sem;      /* semaphore object handle */
<b>Return Value</b>	Void

<b>Assembly Interface</b>	none
---------------------------	------

<b>Description</b>	SEM_ipost readies the first task waiting for the semaphore. If no task is waiting, SEM_ipost simply increments the semaphore count and returns.
--------------------	---

SEM\_ipost is the same as SEM\_post in the DSP/BIOS environment. SEM\_ipost is provided for source compatibility reasons only. For portable code, use SEM\_ipost within an HWI or SWI and SEM\_post within a task.

**Constraints and Calling Context**

- ❑ When called within an HWI ISR, the code sequence calling SEM\_ipost must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.
- ❑ SEM\_ipost cannot be called from the program's main function.

<b>See Also</b>	SEM_pend SEM_post
-----------------	----------------------

SEM\_new

Initialize semaphore object

C Interface

Syntax	Void SEM_new(sem, count);
Parameters	SEM_Handle sem;       /* pointer to semaphore object */ Int                    count;   /* initial semaphore count */
Return Value	Void
Assembly Interface	none
Description	SEM_new initializes the semaphore object pointed to by sem with count. The function should be used on a statically created semaphore for initialization purposes only. No task switch occurs when calling SEM_new.
Constraints and Calling Context	<div><input type="checkbox"/> count must be greater than or equal to 0</div> <div><input type="checkbox"/> no tasks should be pending on the semaphore when SEM_new is called</div>
See Also	QUE_new

**SEM\_pend***Wait for a semaphore***C Interface**

<b>Syntax</b>	status = SEM_pend(sem, timeout);
<b>Parameters</b>	SEM_Handle sem;     /* semaphore object handle */ Uns                    timeout;   /* return after this many system clock ticks */
<b>Return Value</b>	Bool                    status;     /* TRUE if successful, FALSE if timeout */

**Assembly Interface**

none

**Description**

If the semaphore count is greater than zero, SEM\_pend decrements the count and returns TRUE. Otherwise, SEM\_pend suspends the execution of the current task until SEM\_post is called or the timeout expires. If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout is SYS\_FOREVER, the task remains suspended until SEM\_post is called on this semaphore. If timeout is 0, SEM\_pend returns immediately.

If timeout expires (or timeout is 0) before the semaphore is available, SEM\_pend returns FALSE. Otherwise SEM\_pend returns TRUE.

A task switch occurs when calling SEM\_pend if the semaphore count is 0 and timeout is not zero.

**Constraints and Calling Context**

- ❑ SEM\_pend can only be called from an HWI or SWI if timeout is 0.
- ❑ SEM\_pend cannot be called from the program's main function.
- ❑ SEM\_pend can only be called from within a TSK\_disable / TSK\_enable block if timeout is 0.
- ❑ SEM\_pend should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.

**See Also**

SEM\_post

**SEM\_post**

*Signal a semaphore*

**C Interface**

<b>Syntax</b>	SEM_post(sem);
<b>Parameters</b>	SEM_Handle sem;      /* semaphore object handle */
<b>Return Value</b>	Void

**Assembly Interface**      none

**Description**      SEM\_post readies the first task waiting for the semaphore. If no task is waiting, SEM\_post simply increments the semaphore count and returns.

A task switch occurs when calling SEM\_post if a higher priority task is made ready to run.

- Constraints and Calling Context**
- ❑ When called within an HWI ISR, the code sequence calling SEM\_post must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.
  - ❑ SEM\_post cannot be called from within a TSK\_disable/TSK\_enable block.

**See Also**      SEM\_ipost  
SEM\_pend

**SEM\_reset***Reset semaphore count***C Interface****Syntax**`SEM_reset(sem, count);`**Parameters**`SEM_Handle sem;     /* semaphore object handle */  
Int           count;   /* semaphore count */`**Return Value**

Void

**Assembly Interface**

none

**Description**

SEM\_reset resets the semaphore count to count.

No task switch occurs when calling SEM\_reset.

**Constraints and  
Calling Context**

- ❑ count must be greater than or equal to 0.
- ❑ No tasks should be waiting on the semaphore when SEM\_reset is called.
- ❑ SEM\_reset cannot be called by an HWI or a SWI.

**See Also**

SEM\_create

## 2.19 SIO Module

The SIO module is the stream input and output manager.

### Functions

- ❑ SIO\_bufsize. Size of the buffers used by a stream
- ❑ SIO\_create. Create stream
- ❑ SIO\_ctrl. Perform a device-dependent control operation
- ❑ SIO\_delete. Delete stream
- ❑ SIO\_flush. Idle a stream by flushing buffers
- ❑ SIO\_get. Get buffer from stream
- ❑ SIO\_idle. Idle a stream
- ❑ SIO\_issue. Send a buffer to a stream
- ❑ SIO\_put. Put buffer to a stream
- ❑ SIO\_reclaim. Request a buffer back from a stream
- ❑ SIO\_segid. Memory segment used by a stream
- ❑ SIO\_select. Select a ready device
- ❑ SIO\_staticbuf. Acquire static buffer from stream

### Constants, Types, and Structures

```
#define SIO_STANDARD      0 /* open stream for */
                          /* standard streaming model */
#define SIO_ISSUERECLAIM 1 /* open stream for */
                          /* issue/reclaim streaming model */

#define SIO_INPUT        0 /* open for input */
#define SIO_OUTPUT       1 /* open for output */
typedef SIO_Handle;      /* stream object handle */

struct SIO_Attrs {
    Int      nbufs;      /* stream attributes */
    Int      segid;      /* number of buffers */
    Int      align;      /* buffer segment ID */
    Bool     flush;      /* buffer alignment */
    Uns      model;      /* TRUE = don't block in DEV_idle */
    Uns      timeout;    /* usage model: */
                      /* SIO_STANDARD/SIO_ISSUERECLAIM */
};

SIO_Attrs SIO_ATTRS = {
    2,              /* nbufs */
    0,              /* segid */
    0,              /* align */
    FALSE,          /* flush */
    SIO_STANDARD,   /* model */
    SYS_FOREVER     /* timeout */
};
```

## Description

The stream manager provides efficient real-time device-independent I/O through a set of functions that manipulate stream objects accessed through handles of type `SIO_Handle`. The device independence is afforded by having a common high-level abstraction appropriate for real-time applications, continuous streams of data, that can be associated with a variety of devices. All I/O programming is done in a high-level manner using these stream handles to the devices and the stream manager takes care of dispatching into the underlying device drivers.

For efficiency, streams are treated as sequences of fixed-size buffers of data rather than just sequences of MADUs.

Streams can be opened and closed at any point during program execution using the functions `SIO_create` and `SIO_delete`, respectively.

The `SIO_issue` and `SIO_reclaim` function calls are enhancements to the basic DSP/BIOS device model. These functions provide a second usage model for streaming, referred to as the issue/reclaim model. It is a more flexible streaming model that allows clients to supply their own buffers to a stream, and to get them back in the order that they were submitted. The `SIO_issue` and `SIO_reclaim` functions also provide a user argument that can be used for passing information between the stream client and the stream devices.

## SIO Manager Properties

The following global properties can be set for the SIO module on the SIO Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the SIO objects created with the Configuration Tool

## SIO Object Properties

The following properties can be set for an SIO object on the SIO Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this SIO object.
- ☐ **Device.** Select the device to which you want to bind this SIO object. User-defined devices are listed along with DGN and DPI devices.
- ☐ **Device Control Parameter.** Type the device suffix to be passed to any devices stacked below the device connected to this stream.
- ☐ **Mode.** Select input if this stream is to be used for input to the application program and output if this stream is to be used for output.
- ☐ **Buffer size.** If this stream uses the Standard model, this property controls the size of buffers (in MADUs) allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle buffers of any size.



- ☐ **Number of buffers.** If this stream uses the Standard model, this property controls the number of buffers allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle up to the specified Number of buffers.
- ☐ **Place buffers in memory segment.** Select the memory segment to contain the stream buffers if Model is Standard.
- ☐ **Buffer alignment.** Specify the memory alignment to use for stream buffers if Model is Standard. For example, if you select 16, the buffer must begin at an address that is a multiple of 16. The default is 1, which means the buffer can begin at any address.
- ☐ **Flush.** Check this box if you want the stream to discard all pending data and return without blocking if this object is idled at run-time with SIO\_idle.
- ☐ **Model.** Select Standard if you want all buffers to be allocated when the stream is created. Select Issue/Reclaim if your program is to allocate the buffers and supply them using SIO\_issue.
- ☐ **Allocate Static Buffer(s).** If this box is checked, the Configuration Tool allocates stream buffers for the user. The SIO\_staticbuf function is used to acquire these buffers from the stream. When the Standard model is used, checking this box causes one buffer more than the Number of buffers property to be allocated. When the Issue/Reclaim model is used, buffers are not normally allocated. Checking this box causes the number of buffers specified by the Number of buffers property to be allocated.
- ☐ **Timeout for I/O operation.** This parameter specifies the length of time the I/O operations SIO\_get, SIO\_put, and SIO\_reclaim wait for I/O. The device driver's Dxx\_reclaim function typically uses this timeout while waiting for I/O. If the timeout expires before a buffer is available, the I/O operation returns (-1 \* SYS\_ETIMEOUT) and no buffer is returned.

**SIO\_bufsize***Return the size of the buffers used by a stream***C Interface****Syntax**                      size = SIO\_bufsize(stream);**Parameters**                SIO\_Handle stream;**Return Value**              Uns              size;**Assembly Interface**

none

**Description**                SIO\_bufsize returns the size of the buffers used by stream.**See Also**                    SIO\_segid

SIO\_create

Open a stream

C Interface

Syntax	stream = SIO_create(name, mode, bufsize, attrs);
Parameters	<div>String        name;     /* name of device */</div> <div>Int           mode;     /* SIO_INPUT or SIO_OUTPUT */</div> <div>Uns          bufsize;  /* stream buffer size */</div> <div>SIO_Attrs    *attrs;    /* pointer to stream attributes */</div>
Return Value	SIO_Handle stream;   /* stream object handle */
Assembly Interface	none
Description	<p>SIO_create creates a new stream object and opens the device specified by name. If successful, SIO_create returns the handle of the new stream object. If unsuccessful, SIO_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).</p> <p>Internally, SIO_create calls Dxx_open to open a device.</p> <p>The mode parameter specifies whether the stream is to be used for input (SIO_INPUT) or output (SIO_OUTPUT).</p> <p>If the stream is being opened in SIO_STANDARD mode, SIO_create allocates buffers of size bufsize for use by the stream. Initially these buffers are placed on the device todevice queue for input streams, and the device fromdevice queue for output streams.</p> <p>If the stream is being opened in SIO_ISSUERECLAIM mode, SIO_create does not allocate any buffers for the stream. In SIO_ISSUERECLAIM mode all buffers must be supplied by the client via the SIO_issue call. It does, however, prepare the stream for a maximum number of buffers of the specified size.</p> <p>If the attrs parameter is NULL, the new stream is assigned the default set of attributes specified by SIO_ATTRS. The following stream attributes are currently supported:</p>

```

struct SIO_Attrs {
    Int     nbufs;
    Int     segid;
    Int     align;
    Bool    flush;
    Uns     model;
    Uns     timeout;
};

```

The `nbufs` attribute specifies the number of buffers allocated by the stream in the `SIO_STANDARD` usage model, or the number of buffers to prepare for in the `SIO_ISSUERECLAIM` usage model. The default value of `nbufs` is 2. In the `SIO_ISSUERECLAIM` usage model, `nbufs` is the maximum number of buffers that can be outstanding (that is, issued but not reclaimed) at any point in time.

The `segid` attribute specifies the memory segment for stream buffers. Use the memory segment names defined using the Configuration Tool. The default value is 0, meaning that buffers are to be allocated from the Segment for DSP/BIOS objects defined for the MEM manager.

The `align` attribute specifies the memory alignment for stream buffers. The default value is 0, meaning that no alignment is needed.

The `flush` attribute indicates the desired behavior for an output stream when it is deleted. If `flush` is `TRUE`, a call to `SIO_delete` causes the stream to discard all pending data and return without blocking. If `flush` is `FALSE`, a call to `SIO_delete` causes the stream to block until all pending data has been processed. The default value is `FALSE`.

The `model` attribute indicates the usage model that is to be used with this stream. The two usage models are `SIO_ISSUERECLAIM` and `SIO_STANDARD`. The default usage model is `SIO_STANDARD`.

The `timeout` attribute specifies the length of time the device driver waits for I/O completion before returning an error (for example, `SYS_ETIMEOUT`). `timeout` is usually passed as a parameter to `SEM_pend` by the device driver. The default is `SYS_FOREVER` which indicates that the driver waits forever. If `timeout` is `SYS_FOREVER`, the task remains suspended until a buffer is available to be returned by the stream. The `timeout` attribute applies to the I/O operations `SIO_get`, `SIO_put`, and `SIO_reclaim`. If `timeout` is 0, the I/O operation returns immediately. If the `timeout` expires before a buffer is available to be returned, the I/O operation returns the value of `(-1 * SYS_ETIMEOUT)`. Otherwise the I/O operation returns the number of valid MADUs in the buffer, or -1 multiplied by an error code.

SIO\_create calls MEM\_alloc to dynamically create the object's data structure. MEM\_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2-136.

### **Constraints and Calling Context**

- ❑ A stream can only be used by one task simultaneously. Catastrophic failure can result if more than one task calls SIO\_get (or SIO\_issue / SIO\_reclaim) on the same input stream, or more than one task calls SIO\_put (or SIO\_issue / SIO\_reclaim) on the same output stream.
- ❑ SIO\_create creates a stream dynamically. Do not call SIO\_create on a stream that was created with the Configuration Tool.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions. However, streams that are to be used with stacking drivers must be created dynamically with SIO\_create.
- ❑ SIO\_create cannot be called from an SWI or HWI.

### **See Also**

Dxx\_open  
MEM\_alloc  
SEM\_pend  
SIO\_delete  
SIO\_issue  
SIO\_reclaim  
SYS\_error

**SIO\_ctrl***Perform a device-dependent control operation***C Interface****Syntax**

```
status = SIO_ctrl(stream, cmd, arg);
```

**Parameters**

```
SIO_Handle stream; /* stream handle */  
Uns      cmd;     /* command to device */  
Arg      arg;     /* arbitrary argument */
```

**Return Value**

```
Int      status; /* device status */
```

**Assembly Interface**

none

**Description**

SIO\_ctrl causes a control operation to be issued to the device associated with stream. cmd and arg are passed directly to the device.

SIO\_ctrl returns SYS\_OK if successful, and a non-zero device-dependent error value if unsuccessful.

Internally, SIO\_ctrl calls Dxx\_ctrl to send control commands to a device.

**Constraints and  
Calling Context**

- ❑ SIO\_ctrl cannot be called from an SWI or HWI.

**See Also**

Dxx\_ctrl

**SIO\_delete***Close a stream and free its buffers***C Interface**

<b>Syntax</b>	<code>status = SIO_delete(stream);</code>
<b>Parameters</b>	<code>SIO_Handle stream; /* stream object */</code>
<b>Return Value</b>	<code>Int status; /* result of operation */</code>

**Assembly Interface**

none

**Description**

SIO\_delete idles the device before freeing the stream object and buffers.

If the stream being deleted was opened for input, then any pending input data is discarded. If the stream being deleted was opened for output, the method for handling data is determined by the value of the object's Flush property in the Configuration Tool or the flush field in the SIO\_Attrs structure (passed in with SIO\_create). If flush is TRUE, SIO\_delete discards all pending data and return without blocking. If flush is FALSE, SIO\_delete blocks until all pending data has been processed by the stream.

SIO\_delete returns SYS\_OK if and only if the operation is successful.

SIO\_delete calls MEM\_free to delete a stream. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Internally, SIO\_delete first calls Dxx\_idle to idle the device. Then it calls Dxx\_close.

**Constraints and Calling Context**

- ❑ SIO\_delete cannot be called from an SWI or HWI.
- ❑ No check is performed to prevent SIO\_delete from being used on a statically-created object. If a program attempts to delete a stream object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

SIO\_create  
SIO\_flush  
SIO\_idle  
Dxx\_idle  
Dxx\_close

**SIO\_flush***Flush a stream***C Interface**

**Syntax**                      `status = SIO_flush(stream);`

**Parameters**                `SIO_Handle stream;   /* stream handle */`

**Return Value**              `Int                   status;   /* result of operation */`

**Assembly Interface**

none

**Description**

SIO\_flush causes all pending data to be discarded regardless of the mode of the stream. SIO\_flush differs from SIO\_idle in that SIO\_flush never suspends program execution to complete processing of data, even for a stream created in output mode.

The underlying device connected to stream is idled as a result of calling SIO\_flush. In general, the interrupt is disabled for the device.

One of the purposes of this function is to provide synchronization with the external environment.

SIO\_flush returns SYS\_OK if and only if the stream is successfully idled.

Internally, SIO\_flush calls Dxx\_idle and flushes all pending data.

**Constraints and  
Calling Context**

❑ SIO\_flush cannot be called from an SWI or HWI.

**See Also**

Dxx\_idle  
SIO\_create  
SIO\_idle



**SIO\_get***Get a buffer from stream***C Interface**

<b>Syntax</b>	<code>nmadus = SIO_get(stream, bufp);</code>
<b>Parameters</b>	<code>SIO_Handle stream</code> /* stream handle */ <code>Ptr *bufp;</code> /* pointer to a buffer */
<b>Return Value</b>	<code>Int nmadus;</code> /* number of MADUs read or error if negative */

**Assembly Interface**

none

**Description**

SIO\_get exchanges an empty buffer with a non-empty buffer from stream. The bufp parameter is an input/output parameter which points to an empty buffer when SIO\_get is called. When SIO\_get returns, bufp points to a new (different) buffer, and nmadus indicates success or failure of the SIO\_get call.

SIO\_get blocks until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO\_create). If a timeout occurs, the value (-1 \* SYS\_ETIMEOUT) is returned. If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO\_get returns a positive value for nmadus. As a success indicator, nmadus is the number of MADUs received from the stream. To indicate failure, SIO\_get returns a negative value for nmadus. As a failure indicator, nmadus is the actual error code multiplied by -1.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO\_get.

A task switch occurs when calling SIO\_get if there are no non-empty data buffers in stream.

Internally, SIO\_get calls Dxx\_issue and Dxx\_reclaim for the device.

**Constraints and Calling Context**

- ❑ The stream must not be created with attrs.model set to SIO\_ISSUERECLAIM. The results of calling SIO\_get on a stream created for the issue/reclaim streaming model are undefined.
- ❑ SIO\_get cannot be called from an SWI or HWI.

**See Also**

Dxx\_issue  
Dxx\_reclaim  
SIO\_create  
SIO\_put

**SIO\_idle***Idle a stream***C Interface**

**Syntax**                      `status = SIO_idle(stream);`

**Parameters**                `SIO_Handle stream;   /* stream handle */`

**Return Value**              `Int                   status;   /* result of operation */`

**Assembly Interface**

none

**Description**

If stream is being used for output, SIO\_idle causes any currently buffered data to be transferred to the output device associated with stream. SIO\_idle suspends program execution for as long as is required for the data to be consumed by the underlying device.

If stream is being used for input, SIO\_idle causes any currently buffered data to be discarded. The underlying device connected to stream is idled as a result of calling SIO\_idle. In general, the interrupt is disabled for this device.

If discarding of unrendered output is desired, use SIO\_flush instead.

One of the purposes of this function is to provide synchronization with the external environment.

SIO\_idle returns SYS\_OK if and only if the stream is successfully idled.

Internally, SIO\_idle calls Dxx\_idle to idle the device.

**Constraints and Calling Context**

❑ SIO\_idle cannot be called from an SWI or HWI.

**See Also**

Dxx\_idle  
SIO\_create  
SIO\_flush

**SIO\_issue***Send a buffer to a stream***C Interface****Syntax**

```
status = SIO_issue(stream, pbuf, nmadus, arg);
```

**Parameters**

```
SIO_Handle stream; /* stream handle */
Ptr        pbuf;   /* pointer to a buffer */
Uns        nmadus; /* number of MADUs in the buffer */
Arg        arg;    /* user argument */
```

**Return Value**

```
Int        status; /* result of operation */
```

**Assembly Interface**

none

**Description**

SIO\_issue is used to send a buffer and its related information to a stream. The buffer-related information consists of the logical length of the buffer (nmadus), and the user argument to be associated with that buffer. SIO\_issue sends a buffer to the stream and return to the caller without blocking. It also returns an error code indicating success (SYS\_OK) or failure of the call.

Internally, SIO\_issue calls Dxx\_issue after placing a new input frame on the driver's device->todevice queue.

Failure of SIO\_issue indicates that the stream was not able to accept the buffer being issued or that there was a device error when the underlying Dxx\_issue was called. In the first case, the application is probably issuing more frames than the maximum MADUs allowed for the stream, before it reclaims any frames. In the second case, the failure reveals an underlying device driver or hardware problem. If SIO\_issue fails, SIO\_idle should be called for an SIO\_INPUT stream, and SIO\_flush should be called for an SIO\_OUTPUT stream, before attempting more I/O through the stream.

The interpretation of nmadus, the logical size of a buffer, is direction-dependent. For a stream opened in SIO\_OUTPUT mode, the logical size of the buffer indicates the number of valid MADUs of data it contains. For a stream opened in SIO\_INPUT mode, the logical length of a buffer indicates the number of MADUs being requested by the client. In either case, the logical size of the buffer must be less than or equal to the physical size of the buffer.

The argument arg is not interpreted by DSP/BIOS, but is offered as a service to the stream client. DSP/BIOS and all DSP/BIOS-compliant device drivers preserve the value of arg and maintain its association with the data that it was issued with. arg provides a user argument as a method for a client to associate additional information with a particular buffer of data.

SIO\_issue is used in conjunction with SIO\_reclaim to operate a stream opened in SIO\_ISSUERECLAIM mode. The SIO\_issue call sends a buffer to a stream, and SIO\_reclaim retrieves a buffer from a stream. In normal operation each SIO\_issue call is followed by an SIO\_reclaim call. Short bursts of multiple SIO\_issue calls can be made without an intervening SIO\_reclaim call, but over the life of the stream SIO\_issue and SIO\_reclaim must be called the same number of times.

At any given point in the life of a stream, the number of SIO\_issue calls can exceed the number of SIO\_reclaim calls by a maximum of nbufs. The value of nbufs is determined by the SIO\_create call or by setting the Number of buffers property for the object in the Configuration Tool.

---

**Note:**

An SIO\_reclaim call should not be made without at least one outstanding SIO\_issue call. Calling SIO\_reclaim with no outstanding SIO\_issue calls has undefined results.

---

**Constraints and  
Calling Context**

- ❑ The stream must be created with attrs.model set to SIO\_ISSUERECLAIM.
- ❑ SIO\_issue cannot be called from an SWI or HWI.

**See Also**

Dxx\_issue  
SIO\_create  
SIO\_reclaim

**SIO\_put***Put a buffer to a stream***C Interface****Syntax**

```
nmadus = SIO_put(stream, bufp, nmadus);
```

**Parameters**

```
SIO_Handle stream; /* stream handle */
Ptr         *bufp; /* pointer to a buffer */
Uns         nmadus; /* number of MADUs in the buffer */
```

**Return Value**

```
Int         nmadus; /* number of MADUs, negative if error */
```

**Assembly Interface**

```
none
```

**Description**

SIO\_put exchanges a non-empty buffer with an empty buffer. The bufp parameter is an input/output parameter that points to a non-empty buffer when SIO\_put is called. When SIO\_put returns, bufp points to a new (different) buffer, and nmadus indicates success or failure of the call.

SIO\_put blocks until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO\_create). If a timeout occurs, the value (-1 \* SYS\_ETIMEOUT) is returned. If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO\_put returns a positive value for nmadus. As a success indicator, nmadus is the number of valid MADUs in the buffer returned by the stream (usually zero). To indicate failure, SIO\_put returns a negative value (the actual error code multiplied by -1).

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO\_put.

A task switch occurs when calling SIO\_put if there are no empty data buffers in the stream.

Internally, SIO\_put calls Dxx\_issue and Dxx\_reclaim for the device.

**Constraints and Calling Context**

- ❑ The stream must not be created with attrs.model set to SIO\_ISSUERECLAIM. The results of calling SIO\_put on a stream created for the issue/reclaim model are undefined.
- ❑ SIO\_put cannot be called from an SWI or HWI.

**See Also**

```
Dxx_issue
Dxx_reclaim
SIO_create
SIO_get
```

**SIO\_reclaim***Request a buffer back from a stream***C Interface**

**Syntax** `nmadus = SIO_reclaim(stream, pbufp, parg);`

**Parameters**

<code>SIO_Handle</code>	<code>stream;</code>	<code>/* stream handle */</code>
<code>Ptr</code>	<code>*pbufp;</code>	<code>/* pointer to the buffer */</code>
<code>Arg</code>	<code>*parg;</code>	<code>/* pointer to a user argument */</code>

**Return Value** `Int` `nmadus;` `/* number of MADUs or error if negative */`

**Assembly Interface** `none`

**Description** SIO\_reclaim is used to request a buffer back from a stream. It returns a pointer to the buffer, the number of valid MADUs in the buffer, and a user argument (parg). After the SIO\_reclaim call parg points to the same value that was passed in with this buffer using the SIO\_issue call.

Internally, SIO\_reclaim calls Dxx\_reclaim, then it gets the frame from the driver's device->fromdevice queue.

If the stream was created in SIO\_OUTPUT mode, then SIO\_reclaim returns an empty buffer, and nmadus is zero, since the buffer is empty. If the stream was opened in SIO\_INPUT mode, SIO\_reclaim returns a non-empty buffer, and nmadus is the number of valid MADUs of data in the buffer. In either mode SIO\_reclaim blocks until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO\_create), and it returns a positive number or zero (indicating success), or a negative number (indicating an error condition). If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO\_reclaim returns a positive value for nmadus. As a success indicator, nmadus is the number of valid MADUs in the buffer. To indicate failure, SIO\_reclaim returns a negative value for nmadus. As a failure indicator, nmadus is the actual error code multiplied by -1.

Failure of SIO\_reclaim indicates that no buffer was returned to the client. Therefore, if SIO\_reclaim fails, the client should not attempt to de-reference pbufp, since it is not guaranteed to contain a valid buffer pointer.

SIO\_reclaim is used in conjunction with SIO\_issue to operate a stream opened in SIO\_ISSUERECLAIM mode. The SIO\_issue call sends a buffer to a stream, and SIO\_reclaim retrieves a buffer from a stream. In normal operation each SIO\_issue call is followed by an SIO\_reclaim call. Short bursts of multiple SIO\_issue calls can be made without an intervening SIO\_reclaim call, but over the life of the stream SIO\_issue and SIO\_reclaim must be called the same number of times. The number of SIO\_issue calls can exceed the number of SIO\_reclaim calls by a maximum of nbufs at any given time. The value of nbufs is determined by the SIO\_create call or by setting the Number of buffers property for the object in the Configuration Tool.

---

**Note:**

An SIO\_reclaim call should not be made without at least one outstanding SIO\_issue call. Calling SIO\_reclaim with no outstanding SIO\_issue calls has undefined results.

---

SIO\_reclaim only returns buffers that were passed in using SIO\_issue. It also returns the buffers in the same order that they were issued.

A task switch occurs when calling SIO\_reclaim if timeout is not set to 0, and there are no data buffers available to be returned.

**Constraints and  
Calling Context**

- ☐ The stream must be created with attrs.model set to SIO\_ISSUERECLAIM.
- ☐ There must be at least one outstanding SIO\_issue when an SIO\_reclaim call is made.
- ☐ All frames issued to a stream must be reclaimed before closing the stream.
- ☐ SIO\_reclaim cannot be called from an SWI or HWI.

**See Also**

Dxx\_reclaim  
SIO\_issue  
SIO\_create

**SIO\_segid***Return the memory segment used by the stream***C Interface**

<b>Syntax</b>	<code>segid = SIO_segid(stream);</code>
<b>Parameters</b>	<code>SIO_Handle stream;</code>
<b>Return Value</b>	<code>Int segid; /* memory segment ID */</code>

<b>Assembly Interface</b>	none
---------------------------	------

<b>Description</b>	SIO_segid returns the identifier of the memory segment that stream uses for buffers.
--------------------	--

<b>See Also</b>	SIO_bufsize
-----------------	-------------



<b>SIO_select</b>	<i>Select a ready device</i>
<b>C Interface</b>	
<b>Syntax</b>	<code>mask = SIO_select(streamtab, nstreams, timeout);</code>
<b>Parameters</b>	<code>SIO_Handle streamtab; /* stream table */</code> <code>Int nstreams; /* number of streams */</code> <code>Uns timeout; /* return after this many system clock ticks */</code>
<b>Return Value</b>	<code>Uns mask; /* stream ready mask */</code>
<b>Assembly Interface</b>	none
<b>Description</b>	<p>SIO_select waits until one or more of the streams in the streamtab[] array is ready for I/O (that is, it does not block when an I/O operation is attempted).</p> <p>streamtab[] is an array of streams where nstreams &lt; 16. The timeout parameter indicates the number of system clock ticks to wait before a stream becomes ready. If timeout is 0, SIO_select returns immediately. If timeout is SYS_FOREVER, SIO_select waits until one of the streams is ready. Otherwise, SIO_select waits for up to 1 system clock tick less than timeout due to granularity in system timekeeping.</p> <p>The return value is a mask indicating which streams are ready for I/O. A 1 in bit position j indicates the stream streamtab[j] is ready.</p> <p>SIO_select results in a context switch if no streams are ready for I/O.</p> <p>Internally, SIO_select calls Dxx_ready to determine if the device is ready for an I/O operation.</p>
<b>Constraints and Calling Context</b>	<ul style="list-style-type: none"><li>❑ streamtab must contain handles of type SIO_Handle returned from prior calls to SIO_create.</li><li>❑ streamtab[] is an array of streams; streamtab[i] corresponds to bit position i in mask.</li><li>❑ SIO_select cannot be called from an SWI or HWI.</li></ul>
<b>See Also</b>	Dxx_ready SIO_get SIO_put SIO_reclaim

**SIO\_staticbuf***Acquire static buffer from stream***C Interface**

**Syntax** `nmadus = SIO_staticbuf(stream, bufp);`

**Parameters** `SIO_Handle stream; /* stream handle */`  
`Ptr *bufp; /* pointer to a buffer */`

**Return Value** `Int nmadus; /* number of MADUs in buffer */`

**Assembly Interface**

none

**Description**

SIO\_staticbuf returns buffers for static streams that were configured using the Configuration Tool. Buffers are allocated for static streams by checking the Allocate Static Buffer(s) check box for the related SIO object.

SIO\_staticbuf returns the size of the buffer or 0 if no more buffers are available from the stream.

SIO\_staticbuf can be called multiple times for SIO\_ISSUERECLAIM model streams.

SIO\_staticbuf must be called to acquire all static buffers before calling SIO\_get, SIO\_put, SIO\_issue or SIO\_reclaim.

**Constraints and Calling Context**

- ☐ SIO\_staticbuf should only be called for streams that are defined statically using the Configuration Tool.
- ☐ SIO\_staticbuf should only be called for static streams whose Allocate Static Buffer(s) check box has been checked.
- ☐ SIO\_staticbuf cannot be called after SIO\_get, SIO\_put, SIO\_issue or SIO\_reclaim have been called for the given stream.
- ☐ SIO\_staticbuf cannot be called from an SWI or HWI.

**See Also**

SIO\_get

## 2.20 STS Module

### Functions

The STS module is the statistics objects manager.

- ❑ **STS\_add.** Update statistics using provided value
- ❑ **STS\_delta.** Update statistics using difference between provided value and setpoint
- ❑ **STS\_reset.** Reset values stored in STS object
- ❑ **STS\_set.** Save a setpoint value

### Constants, Types, and Structures



```
struct STS_Obj {
    LgInt    num;      /* count */
    LgInt    acc;      /* total value */
    LgInt    max;      /* maximum value */
}
typedef struct STS_Obj {
    Int      numh;
    Int      numl;
    Int      acch;
    Int      accl;
    Int      maxh;
    Int      maxl;
} STS_Obj;
```

#### Note:

STS objects should not be shared across threads. Therefore, STS\_add, STS\_delta, STS\_reset, and STS\_set are not reentrant.

### Description

The STS module manages objects called statistics accumulators. 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 sum of the individual data values in this series
- ❑ **Maximum.** The largest value already encountered in this series

Using the count and total, the Statistics View analysis tool calculates the average on the host.

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.

## Default STS Tracing

In the RTA Control Panel, you can enable statistics tracing for the following modules by marking the appropriate checkbox. You can also set the HWI object properties to perform various STS operations on registers, addresses, or pointers.

Except for tracing TSK execution, your program does not need to include any calls to STS functions in order to gather these statistics. The default units for the statistics values are shown in Table 2-5.

*Table 2-5. Statistics Units for HWI, PIP, PRD, and SWI Modules*

Module	Units
HWI	Gather statistics on monitored values within HWIs
PIP	Number of frames read from or written to data pipe (count only)
PRD	Number of ticks elapsed from time that the PRD object is ready to run to end of execution
SWI	Instruction cycles elapsed from time posted to completion
TSK	Instruction cycles elapsed from time TSK is made ready to run until the application calls TSK_deltatime

## Custom STS Objects

You can create custom STS objects using the Configuration Tool. The STS\_add operation updates the count, total, and maximum using the value you provide. The STS\_set operation sets a previous value. The STS\_delta operation accumulates the difference between the value you pass and the previous value and updates the previous value to the value you pass.

By using custom STS objects and the STS operations, you can do the following:

- ☐ **Count the number of occurrences of an event.** You can pass a value of 0 to STS\_add. The count statistic tracks how many times your program calls STS\_add for this STS object.
- ☐ **Track the maximum and average values for a variable in your program.** For example, suppose you pass amplitude values to STS\_add. The count tracks how many times your program calls STS\_add for this STS object. The total is the sum of all the amplitudes. The maximum is the largest value. The Statistics View calculates the average amplitude.
- ☐ **Track the minimum value for a variable in your program.** Negate the values you are monitoring and pass them to STS\_add. The maximum is the negative of the minimum value.

- ❑ **Time events or monitor incremental differences in a value.** For example, suppose you want to measure the time between hardware interrupts. You would call STS\_set when the program begins running and STS\_delta each time the interrupt routine runs, passing the result of CLK\_gettime each time. STS\_delta subtracts the previous value from the current value. The count tracks how many times the interrupt routine was performed. The maximum is the largest number of clock counts between interrupt routines. The Statistics View also calculates the average number of clock counts.
- ❑ **Monitor differences between actual values and desired values.** For example, suppose you want to make sure a value stays within a certain range. Subtract the midpoint of the range from the value and pass the absolute value of the result to STS\_add. The count tracks how many times your program calls STS\_add for this STS object. The total is the sum of all deviations from the middle of the range. The maximum is the largest deviation. The Statistics View calculates the average deviation.

You can further customize the statistics data by setting the STS object properties to apply a printf format to the Total, Max, and Average fields in the Statistics View window and choosing a formula to apply to the data values on the host.

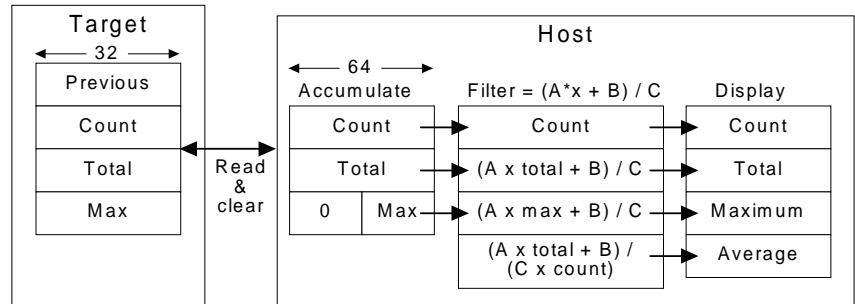
### **Statistics Data Gathering by the Statistics View Analysis Tool**

The statistics manager allows the creation of any number of statistics objects, which in turn can be used by the application to accumulate simple statistics about a time series. This information includes the 32-bit maximum value, the last 32-bit value passed to the object, the number of samples (up to  $2^{32} - 1$  samples), and the 32-bit sum of all samples.

These statistics are accumulated on the target in real-time until the host reads and clears these values on the target. The host, however, continues to accumulate the values read from the target in a host buffer which is displayed by the Statistics View real-time analysis tool. Provided that the host reads and clears the target statistics objects faster than the target can overflow the 32-bit wide values being accumulated, no information loss occurs.

Using the Configuration Tool, you can select a Host Operation for an STS object. The statistics are filtered on the host using the operation and variables you specify. Figure 2-4 shows the effects of the  $(A \times X + B) / C$  operation.

Figure 2-4. Statistics Accumulation on the Host



### STS Manager Properties

The following global property can be set for the STS module on the STS Manager Properties dialog in the Configuration Tool:

- ☐ **Object Memory.** The memory segment that contains the STS objects.

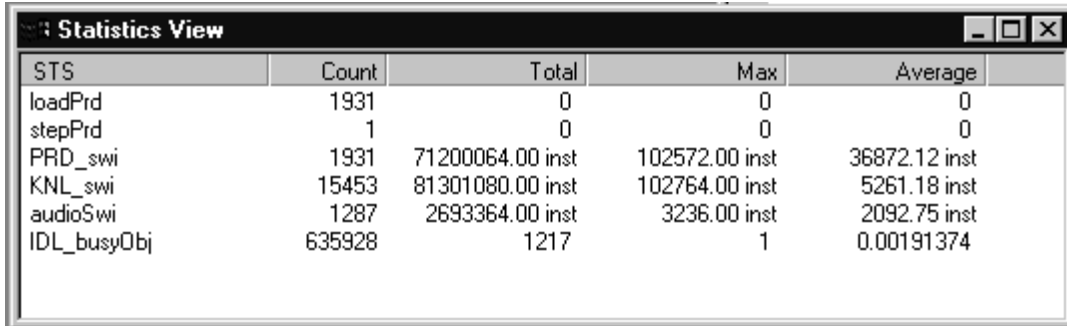
### STS Object Properties

The following properties can be set for a statistics object on the STS Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this STS object
- ☐ **prev.** The initial 32-bit history value to use in this object
- ☐ **unit type.** The unit type property enables you to choose the type of time base units.
  - Not time based. When you select this unit type, the values are displayed in the Statistics View without applying any conversion.
  - High-resolution time based. When you select this unit type, the Statistics View, by default, presents the results in units of instruction cycles.
  - Low-resolution time based. When you select this unit type, the Statistics View, by default, presents the results in units of timer interrupts.
- ☐ **host operation.** The expression evaluated (by the host) on the data for this object before it is displayed by the Statistics View real-time analysis tool. The operation can be:
  - $A \times X$
  - $A \times X + B$
  - $(A \times X + B) / C$
- ☐ **A, B, C.** The integer parameters used by the expression specified by the Host Operation field above.

## STS - Statistics View Interface

You can view statistics in real-time with the Statistics View analysis tool by choosing the DSP/BIOS→Statistics View menu item.



STS	Count	Total	Max	Average
loadPrd	1931	0	0	0
stepPrd	1	0	0	0
PRD_swi	1931	71200064.00 inst	102572.00 inst	36872.12 inst
KNL_swi	15453	81301080.00 inst	102764.00 inst	5261.18 inst
audioSwi	1287	2693364.00 inst	3236.00 inst	2092.75 inst
IDL_busyObj	635928	1217	1	0.00191374

By default, the Statistics View displays all STS objects available. To limit the list of STS objects, right-click on the Statistics View and select Property Page from the pop-up menu. This presents a list of all STS objects. Hold down the control key while selecting the STS object that you wish to observe in the Statistics View. To copy data from the Statistics View, right-click on the Statistics View and select Copy from the pop-up menu. This places the window data in tab-delimited format to the clipboard.

### Note: Updating Task Statistics

If TSK\_deltatime is not called by a task, its STS object will never be updated in the Statistics View, even if TSK accumulators are enabled in the RTA Control Panel.

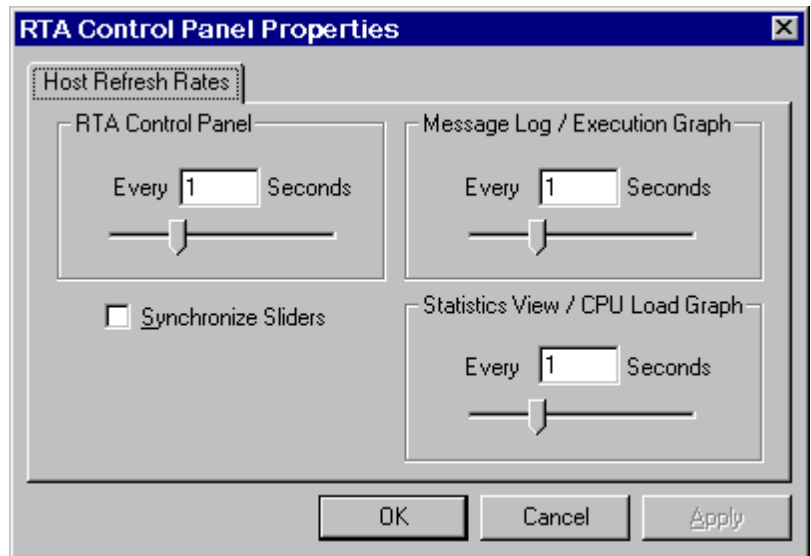
TSK statistics are handled differently than other statistics because TSK functions typically run an infinite loop that blocks when waiting for other threads. In contrast, HWI and SWI functions run to completion without blocking. Because of this difference, DSP/BIOS allows programs to identify the “beginning” of a TSK function’s processing loop by calling TSK\_settime and the “end” of the loop by calling TSK\_deltatime.

To modify the units of time-based STS objects or to provide unit labels for STS objects that are not time based, select the Units tab from the Statistics View Property Page. Select an STS object from the list of STS objects available. The unit options displayed on the right are the unit options for the selected STS object. If the STS object is high-resolution based, you can choose instruction cycles, microseconds, or milliseconds. If your STS object is low-resolution time based, you can choose interrupts, microseconds, or milliseconds. If your STS object is not time based, you can provide a unit label.

When you run your program, the Statistics View displays the Count, Total, Max and Average statistic values for the STS objects. To pause the display, right-click on this window and choose Pause from the pop-up menu. To reset the values to 0, right-click on this window and choose Clear from the pop-up menu.

You can also control how frequently the host polls the target for statistics information. Right-click on the RTA Control Panel and choose the Property Page to set the refresh rate as seen in Figure 2-5. If you set the refresh rate to 0, the host does not poll the target unless you right-click on the Statistics View window and choose Refresh Window from the pop-up menu

Figure 2-5. RTA Control Panel Properties Page



See the *Code Composer Studio* online tutorial for more information on how to monitor statistics with the Statistics View analysis tool.



STS\_add

Update statistics using the provided value

C Interface

Syntax	STS_add(sts, value);
Parameters	STS_Handle sts; /* statistics object handle */ LgInt value; /* new value to update statistics object */
Return Value	Void

Assembly Interface



Syntax	STS_add
Preconditions	ar2 = address of the STS object a = 32-bit value sxm = 1
Postconditions	none
Modifies	ag, ah, al, ar2, bg, bh, bl, c, ovb

Assembly Interface



Syntax	STS_add
Preconditions	xar0 = address of the STS object; ac0 = 32-bit value SXMD = 1
Postconditions	none
Modifies	ac1, xar0

Reentrant no

**Description** STS\_add updates a custom STS object's Total, Count, and Max fields using the data value you provide.

For example, suppose your program passes 32-bit amplitude values to STS\_add. The Count field tracks how many times your program calls

STS\_add for this STS object. The Total field tracks the total of all the amplitudes. The Max field holds the largest value passed to this point. The Statistics View analysis tool calculates the average amplitude.

You can count the occurrences of an event by passing a dummy value (such as 0) to STS\_add and watching the Count field.

You can view the statistics values with the Statistics View analysis tool by enabling statistics in the DSP/BIOS→RTA Control Panel window and choosing your custom STS object in the DSP/BIOS→Statistics View window.

**See Also**

STS\_delta  
STS\_reset  
STS\_set  
TRC\_disable  
TRC\_enable

STS\_delta

Update statistics using the difference between the provided value and the setpoint

C Interface

Syntax	STS_delta(sts,value);
Parameters	STS_Handle sts; /* statistics object handle */ LgInt value; /* new value to update statistics object */
Return Value	Void

Assembly Interface



Syntax	STS_delta
Preconditions	ar2 = address of the STS object a = 32-bit value sxm = 1
Postconditions	none
Modifies	ag, ah, al, ar2, bg, bh, bl, c, ovb

Assembly Interface



Syntax	STS_delta
Preconditions	xar0 = address of the STS object; ac0 = 32-bit value SXMD = 1
Postconditions	none
Modifies	ac1, xar0
Reentrant	no
Description	Each STS object contains a previous value that can be initialized with the Configuration Tool or with a call to STS_set. A call to STS_delta subtracts the previous value from the value it is passed and then invokes STS_add

with the result to update the statistics. STS\_delta also updates the previous value with the value it is passed.

STS\_delta can be used in conjunction with STS\_set to monitor the difference between a variable and a desired value or to benchmark program performance.

You can benchmark your code by using paired calls to STS\_set and STS\_delta that pass the value provided by CLK\_gettime.

```
STS_set(&sts, CLK_gettime());
    "processing to be benchmarked"
STS_delta(&sts, CLK_gettime());
```

## Constraints and Calling Context

- ❑ Before the first call to STS\_delta is made, the previous value of the STS object should be initialized either with a call to STS\_set or by setting the prev property of the STS object using the Configuration Tool.

## Example

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

## See Also

STS\_add  
STS\_reset  
STS\_set  
CLK\_gettime  
CLK\_gettime  
PRD\_getticks  
TRC\_disable  
TRC\_enable

**STS\_reset**

*Reset the values stored in an STS object*

C Interface

Syntax	STS_reset(sts);
Parameters	STS_Handle sts;      /* statistics object handle */
Return Value	Void

Assembly Interface



Syntax	STS_reset
Preconditions	ar2 = address of the STS object
Postconditions	none
Modifies	ag, ah, al, ar2, c

Assembly Interface



Syntax	STS_reset
Preconditions	xar0 = address of the STS object
Postconditions	none
Modifies	xar0, csr

Reentrant	no
-----------	----

Description	<p>STS_reset resets the values stored in an STS object. The Count and Total fields are set to 0 and the Max field is set to the largest negative number. STS_reset does not modify the value set by STS_set.</p> <p>After the Statistics View analysis tool polls statistics data on the target, it performs STS_reset internally. This keeps the 32-bit total and count values from wrapping back to 0 on the target. The host accumulates these values as 64-bit numbers to allow a much larger range than can be stored on the target.</p>
-------------	---

**Example**

```
STS_reset(&sts);  
STS_set(&sts, value);
```

**See Also**

STS\_add  
STS\_delta  
STS\_set  
TRC\_disable  
TRC\_enable

STS\_set

Save a value for STS\_delta

C Interface

Syntax	STS_set(sts, value);
Parameters	STS_Handle sts; /* statistics object handle */ LgInt value; /* new value to update statistics object */
Return Value	Void

Assembly Interface



Syntax	STS_set
Preconditions	ar2 = address of the STS object a = 32-bit value
Postconditions	none
Modifies	none

Assembly Interface



Syntax	STS_set
Preconditions	xar0 = address of the address of the address of the STS object, ac0 = 32-bit value
Postconditions	none
Modifies	none

Reentrant	no
-----------	----

Description	STS_set can be used in conjunction with STS_delta to monitor the difference between a variable and a desired value or to benchmark program performance. STS_set saves a value as the previous value in an STS object. STS_delta subtracts this saved value from the value it is passed and invokes STS_add with the result.
-------------	---

STS\_delta also updates the previous value with the value it was passed. Depending on what you are measuring, you can need to use STS\_set to reset the previous value before the next call to STS\_delta.

You can also set a previous value for an STS object in the Configuration Tool. STS\_set changes this value.

See STS\_delta for details on how to use the value you set with STS\_set.

## Example

This example gathers performance information for the processing between STS\_set and STS\_delta.

```
STS_set(&sts, CLK_gettime());
    "processing to be benchmarked"
STS_delta(&sts, CLK_gettime());
```

This example gathers information about a value's deviation from the desired value.

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

This 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);
STS_set(&sts, baseValue);
```

## See Also

STS\_add  
STS\_delta  
STS\_reset  
TRC\_disable  
TRC\_enable



## 2.21 SWI Module

The SWI module is the software interrupt manager.

### Functions

- ❑ `SWI_andn`. Clear bits from SWI's mailbox; post if becomes 0
- ❑ `SWI_andnHook`. Specialized version of `SWI_andn` for use as hook function for configured DSP/BIOS objects. Both its arguments are of type (Arg).
- ❑ `SWI_create`. Create a software interrupt
- ❑ `SWI_dec`. Decrement SWI's mailbox value; post if becomes 0
- ❑ `SWI_delete`. Delete a software interrupt
- ❑ `SWI_disable`. Disable software interrupts
- ❑ `SWI_enable`. Enable software interrupts
- ❑ `SWI_getattrs`. Get attributes of a software interrupt
- ❑ `SWI_getmbox`. Return an SWI's mailbox value
- ❑ `SWI_getpri`. Return an SWI's priority mask
- ❑ `SWI_inc`. Increment SWI's mailbox value
- ❑ `SWI_or`. Or mask with value contained in SWI's mailbox field
- ❑ `SWI_orHook`. Specialized version of `SWI_or` for use as hook function for configured DSP/BIOS objects. Both its arguments are of type (Arg).
- ❑ `SWI_post`. Post a software interrupt
- ❑ `SWI_raisepri`. Raise an SWI's priority
- ❑ `SWI_restorepri`. Restore an SWI's priority
- ❑ `SWI_self`. Return address of currently executing SWI object
- ❑ `SWI_setattrs`. Set attributes of a software interrupt

### Description

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt service routines.

DSP/BIOS manages four distinct levels of execution threads: hardware interrupt service routines, software interrupts, tasks, and background idle functions. A software interrupt is an object that encapsulates a function to be executed and a priority. Software interrupts are prioritized, preempt tasks, and are preempted by hardware interrupt service routines.

**Note:**

SWI functions are called after the processor register state has been saved. SWI functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

**Note:**

All processor registers are saved before calling SWI functions. This includes st0, st1, a, b, ar0-ar7, the T registers, bk, brc, rsa, rea, and pmst. The following status register bits are set to 0 before calling the user function: ARP, C16, CMPT, CPL, FRCT, BRAF and OVM. If the function is a C function, specified with a leading underscore in the Configuration Tool, CPL is set to 1 before calling the function.

Each software interrupt has a priority level. A software interrupt preempts any lower-priority software interrupt currently executing.

A target program uses an API call to post an SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a software interrupt is posted by an API call, the SWI object's function is not executed immediately. Instead, the function is scheduled for execution. DSP/BIOS uses the software interrupt's priority to determine whether to preempt the thread currently running. Note that if a software interrupt is posted several times before it begins running, because HWIs and higher priority interrupts are running, the software interrupt only runs one time.

Software interrupts can be scheduled for execution with a call to SWI\_post or a number of other SWI functions. Each SWI object has a 16-bit mailbox which is used either to determine whether to post the software interrupt or as a value that can be evaluated within the software interrupt's function. SWI\_andn and SWI\_dec post the software interrupt if the mailbox value transitions to 0. SWI\_or and SWI\_inc also modify the mailbox value. (SWI\_or sets bits, and SWI\_andn clears bits.)

	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\_disable and SWI\_enable operations allow you to post several software interrupts and enable them all for execution at the same time. The software interrupt priorities then determine which software interrupt runs first.

All software interrupts run to completion; you cannot suspend a software interrupt while it waits for something (for example, a device) to be ready. So, you can use the mailbox to tell the software interrupt when all the devices and other conditions it relies on are ready. Within a software interrupt processing function, a call to SWI\_getmbx returns the value of the mailbox when the software interrupt started running. The mailbox is automatically reset to its original value when a software interrupt runs.

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.

A software interrupt preempts any currently running software interrupt with a lower priority. If two software interrupts with the same priority level have been posted, the software interrupt that was posted first runs first. Hardware interrupts in turn preempt any currently running software interrupt, allowing the target to respond quickly to hardware peripherals. For information about setting software interrupt priorities, you can choose Help→Help Topics in the Configuration Tool, click the Index tab, and type priority.

Interrupt threads (including hardware interrupts and software interrupts) are all executed using the same stack. A context switch is performed when a new thread is added to the top of the stack. The SWI module automatically saves the processor's registers before running a higher-priority software interrupt that preempts a lower-priority software interrupt. After the higher-priority software interrupt finishes running, the registers are restored and the lower-priority software interrupt can run if no other higher-priority software interrupts have been posted. (A separate task stack is used by each task thread.)

See the *Code Composer Studio* online tutorial for more information on how to post software interrupts and scheduling issues for the Software Interrupt manager.

## SWI Manager Properties

The following global property can be set for the SWI module on the SWI Manager Properties dialog in the Configuration Tool:

**Object Memory.** The memory segment that contains the SWI objects.

## SWI Object Properties

The following properties can be set for an SWI object on the SWI Object Properties dialog in the Configuration Tool:

- ☐ **comment.** Type a comment to identify this SWI object
- ☐ **function.** The function to execute
- ☐ **priority.** This field shows the numeric priority level for this SWI object. 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. Instead of typing a number in this field, you change the relative priority levels of SWI objects by dragging the objects in the ordered collection view.
- ☐ **mailbox.** The initial value of the 16-bit word used to determine if this software interrupt should be posted.
- ☐ **arg0, arg1.** Two arbitrary pointer type (Arg) arguments to the above configured user function.

## SWI - Code Composer Studio Interface

The SWI tab of the Kernel/Object View shows information about software interrupt objects.

To enable SWI logging, choose DSP/BIOS→RTA Control Panel and put a check in the appropriate box. To view a graph of activity that includes SWI function execution, choose DSP/BIOS→Execution Graph.

You can also enable SWI accumulators in the RTA Control Panel. Then you can choose DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose an SWI object, you see statistics about the number of instruction cycles elapsed from the time the SWI was posted to the SWI function's completion.

---

### Note:

Static SWIs have an STS object associated with them, while dynamic SWIs do not. The STS pointer is located in the SWI object structure for static SWIs only. Therefore, they may be accessed by the user and used for STS operations.

---

**SWI\_andn**

*Clear bits from SWI's mailbox and post if mailbox becomes 0*

**C Interface**

<b>Syntax</b>	SWI_andn(swi, mask);
<b>Parameters</b>	SWI_Handle swi; /* SWI object handle*/ Uns mask /* value to be ANDed */
<b>Return Value</b>	Void

**Assembly Interface**



<b>Syntax</b>	SWI_andn
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE ar2 = address of the SWI object al = mask intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**



<b>Syntax</b>	SWI_andn
<b>Preconditions</b>	xar0 = address of the SWI object t0 = mask intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	yes
<b>Description</b>	SWI_andn is used to conditionally post a software interrupt. SWI_andn clears the bits specified by a mask from SWI's internal mailbox. If SWI's

mailbox becomes 0, SWI\_andn posts the software interrupt. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

For example, if there are multiple conditions that must all be met before a software interrupt can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

SWI\_andn results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

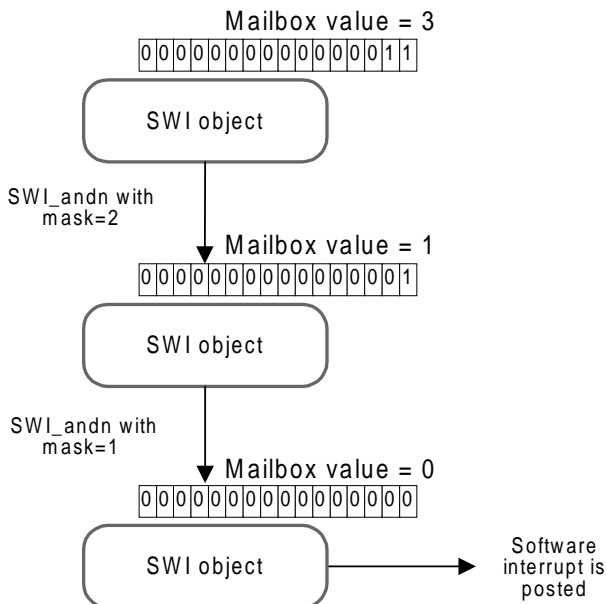
---

**Note:**

Use the specialized version, SWI\_andnHook, when SWI\_andn functionality is required for a DSP/BIOS object hook function.

---

The following figure shows an example of how a mailbox with an initial value of 3 can be cleared by two calls to SWI\_andn with values of 2 and 1. The entire mailbox could also be cleared with a single call to SWI\_andn with a value of 3.



**Constraints and  
Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling SWI\_andn must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**Example**

```
/* ===== ioReady ===== */  
  
Void ioReady(unsigned int mask)  
{  
    /* clear bits of "ready mask" */  
    SWI_andn(&copySWI, mask);  
}
```

**See Also**

SWI\_andnHook  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_orHook  
SWI\_post  
SWI\_self

**SWI\_andnHook***Clear bits from SWI's mailbox and post if mailbox becomes 0***C Interface**

<b>Syntax</b>	SWI_andnHook(swi, mask);		
<b>Parameters</b>	Arg	swi;	/* SWI object handle*/
	Arg	mask	/* value to be ANDed */
<b>Return Value</b>	Void		

**Assembly Interface**

<b>Syntax</b>	SWI_andnHook
<b>Preconditions</b>	cpl = 0
	dp = GBL_A_SYSPAGE
	ar2 = address of the SWI object
	al = mask
	intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_andn
<b>Preconditions</b>	xar0 = address of the SWI object
	t0 = mask
	intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	yes
<b>Description</b>	SWI_andnHook is a specialized version of SWI_andn for use as hook function for configured DSP/BIOS objects. SWI_andnHook clears the bits specified by a mask from SWI's internal mailbox and also moves the



arguments to the correct registers for proper interface with low level DSP/BIOS assembly code. If SWI's mailbox becomes 0, SWI\_andnHook posts the software interrupt. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

For example, if there are multiple conditions that must all be met before a software interrupt can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

SWI\_andnHook results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

### Constraints and Calling Context

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling SWI\_andnHook must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

### Example

```
/* ===== ioReady ===== */  
  
Void ioReady(unsigned int mask)  
{  
    /* clear bits of "ready mask" */  
    SWI_andn(&copySWI, mask);  
}
```

### See Also

SWI\_andn  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_orHook  
SWI\_post  
SWI\_self

**SWI\_create***Create a software interrupt***C Interface**

<b>Syntax</b>	<code>swi = SWI_create(attrs);</code>
<b>Parameters</b>	<code>SWI_Attrs    *attrs;     /* pointer to swi attributes */</code>
<b>Return Value</b>	<code>SWI_Handle swi;         /* handle for new swi object */</code>

**Assembly Interface**

none

**Description**

SWI\_create creates a new SWI object. If successful, SWI\_create returns the handle of the new SWI object. If unsuccessful, SWI\_create returns NULL unless it aborts. For example, SWI\_create can abort if it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the object to be created, facilitates setting the SWI object's attributes. If attrs is NULL, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type SWI\_attrs defined as follows:

```
struct SWI_Attrs {
    SWI_Fxn  fxn;
    Arg      arg0;
    Arg      arg1;
    Bool     iscfxn;
    Int      priority;
    Uns      mailbox;
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The iscfxn attribute must be TRUE if the fxn attribute references a C function (or an assembly function that expects the C run-time environment). This causes the C preconditions to be applied by the SWI scheduler before calling fxn.

The priority attribute specifies the SWI object's execution priority and must range from 0 to 14. 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.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant `SWI_ATTRS`, which can be assigned to a variable of type `SWI_Attrs` prior to calling `SWI_create`.

`SWI_create` calls `MEM_alloc` to dynamically create the object's data structure. `MEM_alloc` must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2–136.

### **Constraints and Calling Context**

- ☐ `SWI_create` cannot be called from an SWI or HWI.
- ☐ The `fxn` attribute cannot be `NULL`.
- ☐ The `priority` attribute must be less than or equal to 14 and greater than or equal to 1.

### **See Also**

`SWI_delete`  
`SWI_getattrs`  
`SWI_setattrs`  
`SYS_error`

**SWI\_dec***Decrement SWI's mailbox value and post if mailbox becomes 0***C Interface**

<b>Syntax</b>	SWI_dec(swi);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_dec
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE ar2 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_dec
<b>Preconditions</b>	xar0 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1

<b>Reentrant</b>	yes
------------------	-----

<b>Description</b>	SWI_dec is used to conditionally post a software interrupt. SWI_dec decrements the value in SWI's mailbox by 1. If SWI's mailbox value becomes 0, SWI_dec posts the software interrupt. You can increment a mailbox value by using SWI_inc, which always posts the software interrupt.
--------------------	--

For example, you would use SWI\_dec if you wanted to post a software interrupt after a number of occurrences of an event.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

SWI\_dec results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

### Constraints and Calling Context

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling SWI\_dec must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

### Example

```
/* ===== strikeOrBall ===== */

Void strikeOrBall(unsigned int call)
{
    if (call == 1) {
        /* initial mailbox value is 3 */
        SWI_dec(&strikeoutSwi);
    }
    if (call == 2) {
        /* initial mailbox value is 4 */
        SWI_dec(&walkSwi);
    }
}
```

### See Also

SWI\_delete  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_post  
SWI\_self

## SWI\_delete *Delete a software interrupt*

### C Interface

<b>Syntax</b>	SWI_delete(swi);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle */
<b>Return Value</b>	Void

**Assembly Interface**      none

**Description**      SWI\_delete uses MEM\_free to free the SWI object referenced by swi.

SWI\_delete calls MEM\_free to delete the SWI object. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

### Constraints and Calling Context

- ❑ swi cannot be the currently executing SWI object (SWI\_self)
- ❑ SWI\_delete cannot be called from an SWI or HWI.
- ❑ SWI\_delete must not be used to delete a statically-created SWI object. No check is performed to prevent SWI\_delete from being used on a statically-created object. If a program attempts to delete a SWI object that was created using the Configuration Tool, SYS\_error is called.

**See Also**      SWI\_create  
                  SWI\_getattrs  
                  SWI\_setattrs  
                  SYS\_error

SWI\_disable

Disable software interrupts

C Interface

Syntax	SWI_disable();
Parameters	Void
Return Value	Void

Assembly Interface



Syntax	SWI_disable
Preconditions	cpl = 0 dp = GBL_A_SYSPAGE
Postconditions	none
Modifies	c

Assembly Interface



Syntax	SWI_disable
Preconditions	intm = 0
Postconditions	none
Modifies	none

Reentrant	yes
-----------	-----

Description	<p>SWI_disable and SWI_enable control SWI software interrupt processing. SWI_disable disables all other SWI functions from running until SWI_enable is called. Hardware interrupts can still run.</p> <p>SWI_disable and SWI_enable allow you to ensure that statements that must be performed together during critical processing are not interrupted. In the following example, the critical section is not preempted by any software interrupts.</p>
-------------	---

```
SWI_disable();  
    `critical section`  
SWI_enable();
```

You can also use SWI\_disable and SWI\_enable to post several software interrupts and allow them to be performed in priority order. See the example that follows.

SWI\_disable calls can be nested. The number of nesting levels is stored internally. Software interrupt handling is not reenabled until SWI\_enable has been called as many times as SWI\_disable.

## Constraints and Calling Context

- ❑ The calls to HWI\_enter and HWI\_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenables software interrupt handling. You should not call SWI\_disable or SWI\_enable within a hardware ISR.
- ❑ SWI\_disable cannot be called from the program's main function.

## Example

```
/* ===== postEm ===== */  
Void postEm  
{  
    SWI_disable();  
  
    SWI_post(&encoderSwi);  
    SWI_andn(&copySwi, mask);  
    SWI_dec(&strikeoutSwi);  
  
    SWI_enable();  
}
```

## See Also

HWI\_disable  
HWI\_enable  
SWI\_enable



SWI\_enable

Enable software interrupts

C Interface

Syntax	SWI_enable();
Parameters	Void
Return Value	Void

Assembly Interface



Syntax	SWI_enable
Preconditions	can only be called if SWI_disable was called before cpl = 0 dp = GBL_A_SYSPAGE
Postconditions	none
Modifies	ag, ah, al, c

Assembly Interface



Syntax	SWI_enable
Preconditions	can only be called if SWI_disable was called before
Postconditions	none
Modifies	tc!, xar1, t0, intm

Reentrant	yes
-----------	-----

Description	<p>SWI_disable and SWI_enable control SWI software interrupt processing. SWI_disable disables all other software interrupt functions from running until SWI_enable is called. Hardware interrupts can still run. See the SWI_disable section for details.</p> <p>SWI_disable calls can be nested. The number of nesting levels is stored internally. Software interrupt handling is not be reenabled until SWI_enable has been called as many times as SWI_disable.</p>
-------------	---

SWI\_enable results in a context switch if a higher-priority SWI is ready to run.

**Constraints and  
Calling Context**

- ❑ The calls to HWI\_enter and HWI\_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenables software interrupt handling. You should not call SWI\_disable or SWI\_enable within a hardware ISR.
- ❑ SWI\_enable cannot be called from the program's main function.

**See Also**

HWI\_disable  
HWI\_enable  
SWI\_disable

**SWI\_getattrs***Get attributes of a software interrupt***C Interface**

<b>Syntax</b>	SWI_getattrs(swi, attrs);
<b>Parameters</b>	SWI_Handle swi;       /* handle of the swi */ SWI_Attrs *attrs;    /* pointer to swi attributes */
<b>Return Value</b>	Void

**Assembly Interface**       none**Description**           SWI\_getattrs retrieves attributes of an existing SWI object.

The swi parameter specifies the address of the SWI object whose attributes are to be retrieved. The attrs parameter, which is the pointer to a structure that contains the retrieved attributes for the SWI object, facilitates retrieval of the attributes of the SWI object.

The SWI object's attributes are specified through a structure of type SWI\_attrs defined as follows:

```
struct SWI_Attrs {  
    SWI_Fxn    fxn;  
    Arg       arg0;  
    Arg       arg1;  
    Int       priority;  
    Uns       mailbox;  
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The priority attribute specifies the SWI object's execution priority and ranges from 0 to 14. 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.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

The following example uses SWI\_getattrs:

```
extern SWI_Handle swi;  
SWI_Attrs attrs;  
  
SWI_getattrs(swi, &attrs);  
attrs.priority = 5;  
SWI_setattrs(swi, &attrs);
```

### Constraints and Calling Context

- ❑ SWI\_getattrs cannot be called from an SWI or HWI.
- ❑ The attrs parameter cannot be NULL.

### See Also

SWI\_create  
SWI\_delete  
SWI\_setattrs

SWI\_getmbox

Return a SWI's mailbox value

C Interface

Syntax	num = Uns SWI_getmbox();
Parameters	Void
Return Value	Uns            num        /* mailbox value */

Assembly Interface



Syntax	SWI_getmbox
Preconditions	cpl = 0 dp = GBL_A_SYSPAGE
Postconditions	al = current software interrupt's mailbox value
Modifies	ag, ah, al, c

Assembly Interface



Syntax	SWI_getmbox
Preconditions	none
Postconditions	t0 = current software interrupt's mailbox value
Modifies	t0

Reentrant            yes

**Description**            SWI\_getmbox returns the value that SWI's mailbox had when the software interrupt started running. DSP/BIOS saves the mailbox value internally so that SWI\_getmbox can access it at any point within an SWI object's function. DSP/BIOS then automatically resets the mailbox to its initial value (defined with the Configuration Tool) so that other threads can continue to use the software interrupt's mailbox.

SWI\_getmbox should only be called within a function run by a SWI object.

The value returned by SWI\_getmbox can be non-zero if the SWI was posted by a call to SWI\_andn or SWI\_dec. Therefore, SWI\_getmbox

provides relevant information only if the SWI was posted by a call to SWI\_or, SWI\_inc, or SWI\_post.

**Constraints and  
Calling Context**

- ❑ SWI\_getmbox cannot be called from an HWI or TSK level.
- ❑ SWI\_getmbox cannot be called from the program's main function.

**Example**

This call could be used within a SWI object's function to use the mailbox value within the function. For example, if you use SWI\_or or SWI\_inc to post a software interrupt, different mailbox values can require different processing.

```
swicount = SWI_getmbox();
```

**See Also**

SWI\_andn  
SWI\_andnHook  
SWI\_dec  
SWI\_inc  
SWI\_or  
SWI\_orHook  
SWI\_post  
SWI\_self

**SWI\_getpri**

*Return a SWI's priority mask*

**C Interface**

<b>Syntax</b>	key = SWI_getpri(swi);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle*/
<b>Return Value</b>	Uns            key      /* Priority mask of swi */

**Assembly Interface**



<b>Syntax</b>	SWI_getpri
<b>Preconditions</b>	ar2 = address of the SWI object
<b>Postconditions</b>	a = SWI object's priority mask
<b>Modifies</b>	ag, ah, al, c

**Assembly Interface**



<b>Syntax</b>	SWI_getpri
<b>Preconditions</b>	xar0 = address of the SWI object
<b>Postconditions</b>	t0 = SWI object's priority mask
<b>Modifies</b>	t0
<b>Reentrant</b>	yes
<b>Description</b>	SWI_getpri returns the priority mask of the SWI passed in as the argument.
<b>Example</b>	<pre>/* Get the priority key of swi1 */ key = SWI_getpri(&amp;swi1);  /* Get the priorities of swi1 and swi3 */ key = SWI_getpri(&amp;swi1)   SWI_getpri(&amp;swi3);</pre>
<b>See Also</b>	SWI_raisepri SWI_restorepri

**SWI\_inc***Increment SWI's mailbox value***C Interface**

<b>Syntax</b>	SWI_inc(swi);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_inc
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE ar2 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_inc
<b>Preconditions</b>	xar0 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	no
<b>Description</b>	SWI_inc increments the value in SWI's mailbox by 1 and posts the software interrupt regardless of the resulting mailbox value. You can decrement a mailbox value by using SWI_dec, which only posts the software interrupt if the mailbox value is 0.



If a software interrupt is posted several times before it has a chance to begin executing, because HWIs and higher priority software interrupts are running, the software interrupt only runs one time. If this situation occurs, you can use SWI\_inc to post the software interrupt. Within the software interrupt's function, you could then use SWI\_getmbox to find out how many times this software interrupt has been posted since the last time it was executed.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI\_getmbox.

SWI\_inc results in a context switch if the SWI is higher priority than the currently executing thread.

### Constraints and Calling Context

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an Hwi ISR, the code sequence calling SWI\_inc must be either wrapped within an Hwi\_enter/Hwi\_exit pair or invoked by the Hwi dispatcher.

### Example

```
extern SWI_ObjMySwi;
/* ===== AddAndProcess ===== */

Void AddAndProcess(int count)
{
    int i;

    for (i = 1; i <= count; ++i)
        SWI_inc(&MySwi);
}
```

### See Also

SWI\_andn  
SWI\_andnHook  
SWI\_dec  
SWI\_getmbox  
SWI\_or  
SWI\_orHook  
SWI\_post  
SWI\_self

**SWI\_or***OR mask with the value contained in SWI's mailbox field***C Interface**

<b>Syntax</b>	SWI_or(swi, mask);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle*/ Uns            mask;    /* value to be ORed */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_or
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE ar2 = address of the SWI object al = mask intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_or
<b>Preconditions</b>	xar0 = address of the SWI object t0 = mask intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	no
<b>Description</b>	SWI_or is used to post a software interrupt. SWI_or sets the bits specified by a mask in SWI's mailbox. SWI_or posts the software interrupt

regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI\_getmbox.

For example, you might use SWI\_or to post a software interrupt if any of three events should cause a software interrupt to be executed, but you want the software interrupt's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

SWI\_or results in a context switch if the SWI is higher priority than the currently executing thread.

---

**Note:**

Use the specialized version, SWI\_orHook, when SWI\_or functionality is required for a DSP/BIOS object hook function.

---

**Constraints and  
Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling SWI\_or must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**See Also**

SWI\_andn  
SWI\_andnHook  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_orHook  
SWI\_post  
SWI\_self

**SWI\_orHook***OR mask with the value contained in SWI's mailbox field***C Interface**

<b>Syntax</b>	SWI_orHook(swi, mask);		
<b>Parameters</b>	Arg	swi;	/* SWI object handle*/
	Arg	mask;	/* value to be ORed */
<b>Return Value</b>	Void		

**Assembly Interface**

<b>Syntax</b>	SWI_orHook
<b>Preconditions</b>	cpl = 0
	dp = GBL_A_SYSPAGE
	ar2 = address of the SWI object
	al = mask
	intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_orHook
<b>Preconditions</b>	xar0 = address of the SWI object
	t0 = mask
	intrm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	no
<b>Description</b>	SWI_orHook is used to post a software interrupt, and should be used when hook functionality is required for DSP/BIOS hook objects. SWI_orHook sets the bits specified by a mask in SWI's mailbox and also

moves the arguments to the correct registers for interfacing with low level DSP/BIOS assembly code. SWI\_orHook posts the software interrupt regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI\_getmbox.

For example, you might use SWI\_orHook to post a software interrupt if any of three events should cause a software interrupt to be executed, but you want the software interrupt's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

SWI\_orHook results in a context switch if the SWI is higher priority than the currently executing thread.

---

**Note:**

Use the specialized version, SWI\_orHook, when SWI\_or functionality is required for a DSP/BIOS object hook function.

---

**Constraints and  
Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling SWI\_orHook must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**See Also**

SWI\_andn  
SWI\_andnHook  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_post  
SWI\_self

**SWI\_post***Post a software interrupt***C Interface**

<b>Syntax</b>	SWI_post(swi);
<b>Parameters</b>	SWI_Handle swi;      /* SWI object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_post
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE ar2 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

**Assembly Interface**

<b>Syntax</b>	SWI_post
<b>Preconditions</b>	xar0 = address of the SWI object intm = 0 (if called outside the context of an ISR)
<b>Postconditions</b>	none
<b>Modifies</b>	tc1, tc2, t0, t1, xar0, xar1, xar2, xar3, xar4, ac0, ac1
<b>Reentrant</b>	no
<b>Description</b>	<p>SWI_post is used to post a software interrupt regardless of the mailbox value. No change is made to the SWI object's mailbox value.</p> <p>To have a PRD object post an SWI object's function, you can set _SWI_post as the function property of a PRD object and the name of the</p>

software interrupt object you want to post its function as the `arg0` property.

`SWI_post` results in a context switch if the SWI is higher priority than the currently executing thread.

### **Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.
- ❑ When called within an HWI ISR, the code sequence calling `SWI_post` must be either wrapped within an `HWI_enter`/`HWI_exit` pair or invoked by the HWI dispatcher.

### **See Also**

`SWI_andn`  
`SWI_dec`  
`SWI_getmbox`  
`SWI_inc`  
`SWI_or`  
`SWI_self`

**SWI\_raisepri***Raise a SWI's priority***C Interface**

<b>Syntax</b>	key = SWI_raisepri(mask);		
<b>Parameters</b>	Uns	mask;	/* mask of desired priority level */
<b>Return Value</b>	Uns	key;	/* key for use with SWI_restorepri */

**Assembly Interface**

<b>Syntax</b>	SWI_raisepri
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE a = priority mask of desired priority level
<b>Postconditions</b>	a = old priority mask
<b>Modifies</b>	ag, ah, al, bg, bh, bl, c

**Assembly Interface**

<b>Syntax</b>	SWI_raisepri
<b>Preconditions</b>	t0 = priority mask of desired priority level
<b>Postconditions</b>	t0 = old priority mask
<b>Modifies</b>	ac0, t0, t1
<b>Reentrant</b>	yes
<b>Description</b>	<p>SWI_raisepri is used to raise the priority of the currently running SWI to the priority mask passed in as the argument.</p> <p>SWI_raisepri can be used in conjunction with SWI_restorepri to provide a mutual exclusion mechanism without disabling software interrupts.</p>



SWI\_raisepri should be called before the shared resource is accessed, and SWI\_restorepri should be called after the access to the shared resource.

A call to SWI\_raisepri not followed by a SWI\_restorepri keeps the SWI's priority for the rest of the processing at the raised level. A SWI\_post of the SWI posts the SWI at its original priority level.

A SWI object's execution priority must range from 0 to 14. 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.

SWI\_raisepri never lowers the current SWI priority.

### **Constraints and Calling Context**

- ❑ SWI\_raisepri cannot be called from an HWI or TSK level.

### **Example**

```
/* raise priority to the priority of swi_1 */
key = SWI_raisepri(SWI_getpri(&swi_1));
--- access shared resource ---
SWI_restore(key);
```

### **See Also**

SWI\_getpri  
SWI\_restorepri

**SWI\_restorepri***Restore a SWI's priority***C Interface**

<b>Syntax</b>	SWI_restorepri(key);
<b>Parameters</b>	Uns            key;        /* key to restore original priority level */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_restorepri
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE a = old priority mask intm = 0 SWI_D_lock < 0 not in an ISR
<b>Postconditions</b>	none
<b>Modifies</b>	ag, ah, al, c, intm, tc

**Assembly Interface**

<b>Syntax</b>	SWI_restorepri
<b>Preconditions</b>	t0 = old priority mask intm = 0 SWI_D_lock < 0 not in an ISR
<b>Postconditions</b>	none
<b>Modifies</b>	t0, intm
<b>Reentrant</b>	yes
<b>Description</b>	SWI_restorepri restores the priority to the SWI's priority prior to the SWI_raisepri call returning the key. SWI_restorepri can be used in

conjunction with `SWI_raisepri` to provide a mutual exclusion mechanism without disabling all software interrupts.

`SWI_raisepri` should be called right before the shared resource is referenced, and `SWI_restorepri` should be called after the reference to the shared resource.

**Constraints and  
Calling Context**

- ❑ `SWI_restorepri` cannot be called from an HWI or TSK level.
- ❑ `SWI_restorepri` cannot be called from the program's main function.

**Example**

```
/* raise priority to the priority of swi_1 */  
key = SWI_raisepri(SWI_getpri(&swi_1));  
--- access shared resource ---  
SWI_restore(key);
```

**See Also**

`SWI_getpri`  
`SWI_raisepri`

**SWI\_self***Return address of currently executing SWI object***C Interface**

<b>Syntax</b>	<code>curswi = SWI_self();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	SWI_Handle swi;      /* handle for current swi object */

**Assembly Interface**

<b>Syntax</b>	SWI_self
<b>Preconditions</b>	cpl = 0 dp = GBL_A_SYSPAGE
<b>Postconditions</b>	al = address of the current SWI object
<b>Modifies</b>	ag, ah, al, c

**Assembly Interface**

<b>Syntax</b>	SWI_self
<b>Preconditions</b>	none
<b>Postconditions</b>	xar0 = address of the current SWI object
<b>Modifies</b>	xar0
<b>Reentrant</b>	yes
<b>Description</b>	SWI_self returns the address of the currently executing software interrupt.
<b>Constraints and Calling Context</b>	<ul style="list-style-type: none"> <li><input type="checkbox"/> SWI_self cannot be called from an HWI or TSK level.</li> <li><input type="checkbox"/> SWI_self cannot be called from the program's main function.</li> </ul>
<b>Example</b>	<p>You can use SWI_self if you want a software interrupt to repost itself:</p> <pre>SWI_post(SWI_self());</pre>

**See Also**

- SWI\_andn
- SWI\_dec
- SWI\_getmbox
- SWI\_inc
- SWI\_or
- SWI\_post

**SWI\_setattrs***Set attributes of a software interrupt***C Interface**

<b>Syntax</b>	<code>SWI_setattrs(swi, attrs);</code>
<b>Parameters</b>	<code>SWI_Handle swi;</code> <i>/* handle of the swi */</i> <code>SWI_Attrs *attrs;</code> <i>/* pointer to swi attributes */</i>
<b>Return Value</b>	<code>Void</code>

**Assembly Interface**      none

**Description**      `SWI_setattrs` sets attributes of an existing SWI object.

The `swi` parameter specifies the address of the SWI object whose attributes are to be set.

The `attrs` parameter, which can be either `NULL` or a pointer to a structure that contains attributes for the SWI object, facilitates setting the attributes of the SWI object. If `attrs` is `NULL`, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type `SWI_attr`s defined as follows:

```
struct SWI_Attrs {
    SWI_Fxn    fxn;
    Arg        arg0;
    Arg        arg1;
    Int        priority;
    Uns        mailbox;
};
```

The `fxn` attribute, which is the address of the swi function, serves as the entry point of the software interrupt service routine.

The `arg0` and `arg1` attributes specify the arguments passed to the swi function, `fxn`.

The `priority` attribute specifies the SWI object's execution priority and must range from 1 to 14. Priority 14 is the highest priority. You cannot use a priority of 0; that priority is reserved for the system SWI that runs the TSK scheduler.

The `mailbox` attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant `SWI_ATTRS`, which can be assigned to a variable of type `SWI_Attrs` prior to calling `SWI_setattrs`.

The following example uses `SWI_setattrs`:

```
extern SWI_Handle swi;  
SWI_Attrs attrs;  
  
SWI_getattrs(swi, &attrs);  
attrs.priority = 5;  
SWI_setattrs(swi, &attrs);
```

### **Constraints and Calling Context**

- ❑ `SWI_setattrs` must not be used to set the attributes of a SWI that is preempted or is ready to run.
- ❑ The `fxn` attribute cannot be `NULL`.
- ❑ The `priority` attribute must be less than or equal to 14 and greater than or equal to 1.

### **See Also**

`SWI_create`  
`SWI_delete`  
`SWI_getattrs`

## 2.22 SYS Module

The SYS modules manages system settings.

### Functions

- ❑ `SYS_abort`. Abort program execution
- ❑ `SYS_atexit`. Stack an exit handler
- ❑ `SYS_error`. Flag error condition
- ❑ `SYS_exit`. Terminate program execution
- ❑ `SYS_printf`. Formatted output
- ❑ `SYS_putchar`. Output a single character
- ❑ `SYS_sprintf`. Formatted output to string buffer
- ❑ `SYS_vprintf`. Formatted output, variable argument list
- ❑ `SYS_vsprintf`. Output formatted data

### Constants, Types, and Structures

```
#define SYS_FOREVER (Uns)-1 /* wait forever */
#define SYS_POLL    (Uns)0  /* don't wait */

#define SYS_OK      0 /* no error */
#define SYS_EALLOC  1 /* memory allocation error */
#define SYS_EFREE   2 /* memory free error */
#define SYS_ENODEV  3 /* device driver not found */
#define SYS_EBUSY   4 /* device driver busy */
#define SYS_EINVAL  5 /* invalid device parameter */
#define SYS_EBADIO  6 /* I/O failure */
#define SYS_EMODE   7 /* bad mode for device driver */
#define SYS_EDOMAIN 8 /* domain error */
#define SYS_ETIMEOUT 9 /* call timed out */
#define SYS_EEOF    10 /* end-of-file */
#define SYS_EDEAD   11 /* previously deleted obj */
#define SYS_EBADOBJ 12 /* invalid object */
#define SYS_EUSER   256 /* user errors start here */

#define SYS_NUMHANDLERS 8 /* # of atexit handlers */

extern String SYS_errors[]; /* array of error strings
*/
```

### Description

The SYS module makes available a set of general-purpose functions that provide basic system services, such as halting program execution and printing formatted text. In general, each SYS function is patterned after a similar function normally found in the standard C library.



SYS does not directly use the services of any other DSP/BIOS module and therefore resides at the bottom of the system. Other DSP/BIOS modules use the services provided by SYS in lieu of similar C library functions. The SYS module provides hooks for binding system-specific code. This allows programs to gain control wherever other DSP/BIOS modules call one of the SYS functions.

## SYS Manager Properties

The following global properties can be set for the SYS module on the SYS Manager Properties dialog in the Configuration Tool:

- ☐ **Trace Buffer Size.** The size of the buffer that contains system trace information. For example, by default the Putc function writes to the trace buffer.
- ☐ **Trace Buffer Memory.** The memory segment that contains system trace information.
- ☐ **Abort Function.** The function to run if the application aborts by calling SYS\_abort. The default function is \_UTL\_doAbort, which logs an error message and calls \_halt.  
If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- ☐ **Error Function.** The function to run if an error flagged by SYS\_error occurs. The default function is \_UTL\_doError, which logs an error message.  
If this function is written in C, use a leading underscore before the C function name.
- ☐ **Exit Function.** The function to run when the application exits by calling SYS\_exit. The default function is UTL\_halt, which loops forever with interrupts disabled and prevents other processing. If this function is written in C, use a leading underscore before the C function name.
- ☐ **Putc Function.** The function to run if the application calls SYS\_putchar, SYS\_printf, or SYS\_vprintf. The default function is \_UTL\_doPutc, which writes a character to the trace buffer.  
If this function is written in C, use a leading underscore before the C function name.

## SYS Object Properties

The SYS module does not support the creation of individual SYS objects.

**SYS\_abort***Abort program execution***C Interface**

<b>Syntax</b>	<code>SYS_abort(format, [arg,] ...);</code>
<b>Parameters</b>	<div>String        format;    /* format specification string */</div> <div>Arg           arg;        /* optional argument */</div>
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

SYS\_abort aborts program execution by calling the function bound to the configuration parameter Abort function, where vars is of type va\_list and represents the sequence of arg parameters originally passed to SYS\_abort.

```
(*(Abort_function))(format, vars)
```

The function bound to Abort function can elect to pass the format and vars parameters directly to SYS\_vprintf or SYS\_vsprintf prior to terminating program execution.

The default Abort function for the SYS manager is \_UTL\_doAbort, which logs an error message and calls UTL\_halt, which is defined in the boot.c file. The UTL\_halt function performs an infinite loop with all processor interrupts disabled.

**Constraints and Calling Context**

- ❑ If the function bound to Abort function is not reentrant, SYS\_abort must be called atomically.

**See Also**

SYS\_exit  
SYS\_printf

<b>SYS_atexit</b>	<i>Stack an exit handler</i>
<b>C Interface</b>	
<b>Syntax</b>	success = SYS_atexit(handler);
<b>Parameters</b>	Fxn            handler    /* exit handler function */
<b>Return Value</b>	Bool            success   /* handler successfully stacked */
<b>Assembly Interface</b>	none
<b>Description</b>	<p>SYS_atexit pushes handler onto an internal stack of functions to be executed when SYS_exit is called. Up to SYS_NUMHANDLERS(8) functions can be specified in this manner. SYS_exit pops the internal stack until empty and calls each function as follows, where status is the parameter passed to SYS_exit:</p> <pre>( *handler )( status )</pre> <p>SYS_atexit returns TRUE if handler has been successfully stacked; FALSE if the internal stack is full.</p> <p>The handlers on the stack are called only if either of the following happens:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> SYS_exit is called.</li><li><input type="checkbox"/> All tasks for which the Don't shut down system while this task is still running property is TRUE have exited. (By default, this includes the TSK_idle task, which manages communication between the target and analysis tools.)</li></ul>
<b>Constraints and Calling Context</b>	<ul style="list-style-type: none"><li><input type="checkbox"/> handler cannot be NULL.</li></ul>

**SYS\_error***Flag error condition***C Interface****Syntax**

SYS\_error(s, errno, [arg], ...);

**Parameters**

String      s;            /\* error string \*/  
 Int         errno;       /\* error code \*/  
 Arg         arg;         /\* optional argument \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SYS\_error is used to flag DSP/BIOS error conditions. Application programs as well as internal functions use SYS\_error to handle program errors.

SYS\_error calls the function bound to Error function to handle errors.

The default Error function for the SYS manager is \_UTL\_doError, which logs an error message, disables interrupts, and then runs in an infinite loop.

**Constraints and Calling Context**

- ❑ The only valid error numbers are the error constants defined in sys.h (SYS\_E\*) or numbers greater than or equal to SYS\_EUSER. Passing any other error values to SYS\_error can cause DSP/BIOS to crash.
- ❑ The string passed to SYS\_error must be non-NULL.

<b>SYS_exit</b>	<i>Terminate program execution</i>
<b>C Interface</b>	
<b>Syntax</b>	SYS_exit(status);
<b>Parameters</b>	Int                    status;    /* termination status code */
<b>Return Value</b>	Void
<b>Assembly Interface</b>	none
<b>Description</b>	<p>SYS_exit first pops a stack of handlers registered through the function SYS_atexit, and then terminates program execution by calling the function bound to the configuration parameter Exit function, passing on its original status parameter.</p> <pre>( *handlerN )( status )     . . . ( *handler2 )( status ) ( *handler1 )( status )  ( *(Exit_function) )( status )</pre> <p>The default Exit function for the SYS manager is UTL_halt, which performs an infinite loop with all processor interrupts disabled.</p>
<b>Constraints and Calling Context</b>	<p>❑ If the function bound to Exit function or any of the handler functions is not reentrant, SYS_exit must be called atomically.</p>
<b>See Also</b>	SYS_abort SYS_atexit

**SYS\_printf***Output formatted data***C Interface****Syntax**

SYS\_printf(format, [arg,] ...);

**Parameters**

String        format;    /\* format specification string \*/  
 Arg         arg;        /\* optional argument \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SYS\_printf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_printf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_printf are limited to the characters shown in Table 2-6.

*Table 2-6. Conversion Characters Recognized by SYS\_printf*

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

`%[-][0][width]type`

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS\_vprintf is equivalent to SYS\_printf, except that the optional set of arguments is replaced by a va\_list on which the standard C macro va\_start has already been applied. SYS\_sprintf and SYS\_vsprintf are counterparts of SYS\_printf and SYS\_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS\_printf and SYS\_vprintf internally call the function SYS\_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

## **Constraints and Calling Context**

- ❑ The function bound to Exit function or any of the handler functions are not reentrant; SYS\_exit must be called atomically.

## **See Also**

SYS\_vprintf  
SYS\_sprintf  
SYS\_vsprintf

**SYS\_sprintf***Output formatted data***C Interface****Syntax**

SYS\_sprintf (buffer, format, [arg,] ...);

**Parameters**

String      buffer;      /\* output buffer \*/  
String      format;      /\* format specification string \*/  
Arg          arg;          /\* optional argument \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SYS\_sprintf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_sprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_sprintf are limited to the characters in Table 2-7.

*Table 2-7. Conversion Characters Recognized by SYS\_sprintf*

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer



Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

`%[-][0][width]type`

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS\_vprintf is equivalent to SYS\_printf, except that the optional set of arguments is replaced by a va\_list on which the standard C macro va\_start has already been applied. SYS\_sprintf and SYS\_vsprintf are counterparts of SYS\_printf and SYS\_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS\_printf and SYS\_vprintf internally call the function SYS\_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

## Constraints and Calling Context

- ❑ The function bound to Exit function or any of the handler functions are not reentrant; SYS\_exit must be called atomically.

## See Also

SYS\_printf  
SYS\_vprintf  
SYS\_vsprintf

**SYS\_vprintf***Output formatted data***C Interface**

<b>Syntax</b>	SYS_vprintf(format, vars);
<b>Parameters</b>	String        format;    /* format specification string */ va_list       vars;     /* variable argument list reference */
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

SYS\_vprintf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_vprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_vprintf are limited to the characters in Table 2-8.

*Table 2-8. Conversion Characters Recognized by SYS\_vprintf*

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

`%[-][0][width]type`

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS\_vprintf is equivalent to SYS\_printf, except that the optional set of arguments is replaced by a va\_list on which the standard C macro va\_start has already been applied. SYS\_sprintf and SYS\_vsprintf are counterparts of SYS\_printf and SYS\_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS\_printf and SYS\_vprintf internally call the function SYS\_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

## Constraints and Calling Context

- ❑ The function bound to Exit function or any of the handler functions are not reentrant; SYS\_exit must be called atomically.

## See Also

SYS\_printf  
SYS\_sprintf  
SYS\_vsprintf

**SYS\_vsprintf**

*Output formatted data*

**C Interface**

<b>Syntax</b>	SYS_vsprintf(buffer, format, vars);
<b>Parameters</b>	String        buffer;    /* output buffer */ String        format;   /* format specification string */ va_list       vars;     /* variable argument list reference */
<b>Return Value</b>	Void

**Assembly Interface**        none

**Description**                SYS\_vsprintf provides a subset of the capabilities found in the standard C library function printf.

---

**Note:**

SYS\_vsprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

---

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_vsprintf are limited to the characters in Table 2-9.

*Table 2-9.    Conversion Characters Recognized by SYS\_vsprintf*

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

`%[-][0][width]type`

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS\_vprintf is equivalent to SYS\_printf, except that the optional set of arguments is replaced by a va\_list on which the standard C macro va\_start has already been applied. SYS\_sprintf and SYS\_vsprintf are counterparts of SYS\_printf and SYS\_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS\_printf and SYS\_vprintf internally call the function SYS\_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

## Constraints and Calling Context

- ❑ The function bound to Exit function or any of the handler functions are not reentrant; SYS\_exit must be called atomically.

## See Also

SYS\_printf  
SYS\_sprintf  
SYS\_vprintf

**SYS\_putchar***Output a single character***C Interface****Syntax**

SYS\_putchar(c);

**Parameters**

Char            c;            /\* next output character \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SYS\_putchar outputs the character c by calling the system-dependent function bound to the configuration parameter Putc function.

```
((Putc function))(c)
```

For systems with limited I/O capabilities, the function bound to Putc function might simply place c into a global buffer that can be examined after program termination.

The default Putc function for the SYS manager is \_UTL\_doPutc, which writes a character to the trace buffer.

SYS\_putchar is also used internally by SYS\_printf and SYS\_vprintf when generating their output.

**Constraints and Calling Context**

- ❑ If the function bound to Putc function is not reentrant, SYS\_putchar must be called atomically.

**See Also**

SYS\_printf

2.23 TRC Module

The TRC module is the trace manager.

- Functions
- ☐ TRC\_disable. Disable trace class(es)
- ☐ TRC\_enable. Enable trace type(s)
- ☐ TRC\_query. Query trace class(es)

Description

The TRC module manages a set of trace control bits which control the real-time capture of program information through event logs and statistics accumulators. For greater efficiency, the target does not store log or statistics information unless tracing is enabled.

Table 2-10 lists events and statistics that can be traced. The constants defined in trc.h, trc.h54, and trc.h55 are shown in the left column.

Table 2-10. Events and Statistics Traced by TRC

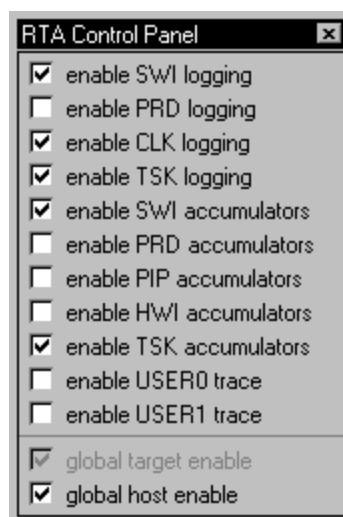
Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Log timer interrupts	off
TRC_LOGPRD	Log periodic ticks and start of periodic functions	off
TRC_LOGSWI	Log events when a software interrupt is posted and completes	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes	off
TRC_STSHWI	Gather statistics on monitored values within HWIs	off
TRC_STSPIP	Count number of frames read from or written to data pipe	off
TRC_STSPRD	Gather statistics on number of ticks elapsed during execution	off
TRC_STSSWI	Gather statistics on length of SWI execution	off
TRC_STSTSK	Gather statistics on length of TSK execution. Statistics are gathered from the time TSK is made ready to run until the application calls TSK_deltatime.	off
TRC_USER0 and TRC_USER1	Your program can use these bits to enable or disable 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.	off
TRC_GBLHOST	This bit must be set in order for any implicit instrumentation to be performed. Simultaneously starts or stops gathering of all enabled types of tracing. This can be important if you are trying to correlate events of different types. This	off
TRC_GBLTARG	This bit must also be set for any implicit instrumentation to be performed. This bit can only be set by the target program and is enabled by default.	on
TRC_STSSWI	Gather statistics on length of SWI execution	off

All trace constants except TRC\_GBLTARG are switched off initially. To enable tracing you can use calls to TRC\_enable or the DSP/BIOS→RTA Control Panel, which uses the TRC module internally. You do not need to enable tracing for messages written with LOG\_printf or LOG\_event and statistics added with STS\_add or STS\_delta.

Your program can call the TRC\_enable and TRC\_disable operations to explicitly start and stop event logging or statistics accumulation in response to conditions encountered during real-time execution. This enables you to preserve the specific log or statistics information you need to see.

## TRC - Code Composer Studio Interface

You can choose DSP/BIOS→RTA Control Panel to open a window that allows you to control run-time tracing.

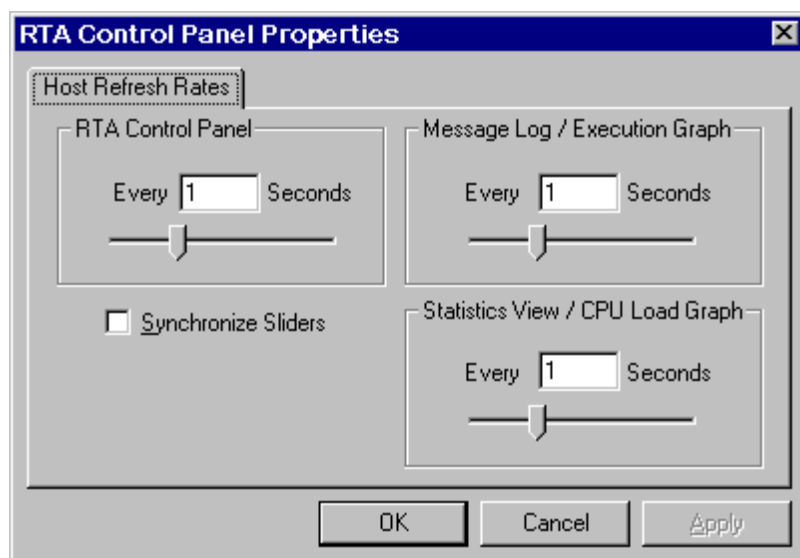


Once you have enabled tracing, you can use DSP/BIOS→Execution Graph and DSP/BIOS→Event Log to see log information, and DSP/BIOS→Statistics View to see statistical information.

You can also control how frequently the host polls the target for trace information. Right-click on the RTA Control Panel and choose the Property Page to set the refresh rate as seen in Figure 2-6. If you set the refresh rate to 0, the host does not poll the target unless you right-click on the RTA Control Panel and choose Refresh Window from the pop-up menu



Figure 2-6. RTA Control Panel Properties Page



See the *Code Composer Studio* online tutorial for more information on how to enable tracing in the RTA Control Panel.

**TRC\_disable***Disable trace class(es)***C Interface**

<b>Syntax</b>	TRC_disable(mask);
<b>Parameters</b>	Uns            mask;     /* trace type constant mask */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	TRC_disable mask
<b>Inputs</b>	mask
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)
<b>Postconditions</b>	none
<b>Modifies</b>	c

**Assembly Interface**

<b>Syntax</b>	TRC_disable mask
<b>Inputs</b>	mask
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)
<b>Postconditions</b>	none
<b>Modifies</b>	none

<b>Reentrant</b>	no
------------------	----

<b>Description</b>	TRC_disable disables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module for a list of constants to use in the mask.)
--------------------	---

The following C code would disable tracing of statistics for software interrupts and periodic functions:

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

Internally, DSP/BIOS uses a bitwise AND NOT operation to disable multiple trace types.

For example, you might want to use TRC\_disable with a circular log and disable tracing when an unwanted condition occurs. This allows test equipment to retrieve the log events that happened just before this condition started.

**See Also**

TRC\_enable  
TRC\_query  
LOG\_printf  
LOG\_event  
STS\_add  
STS\_delta

**TRC\_enable***Enable trace type(s)***C Interface**

<b>Syntax</b>	TRC_enable(mask);
<b>Parameters</b>	Uns            mask;    /* trace type constant mask */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	TRC_enable mask
<b>Inputs</b>	mask
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)
<b>Postconditions</b>	none
<b>Modifies</b>	c

**Assembly Interface**

<b>Syntax</b>	TRC_enable mask
<b>Inputs</b>	mask
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)
<b>Postconditions</b>	none
<b>Modifies</b>	none

<b>Reentrant</b>	no
------------------	----

<b>Description</b>	<p>TRC_enable enables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module for a list of constants to use in the mask.)</p>
--------------------	--

The following C code would enable tracing of statistics for software interrupts and periodic functions:

```
TRC_enable(TRC_STSSWI | TRC_STSPRD);
```

Internally, DSP/BIOS uses a bitwise OR operation to enable multiple trace types.

For example, you might want to use TRC\_enable with a fixed log to enable tracing when a specific condition occurs. This allows test equipment to retrieve the log events that happened just after this condition occurred.

**See Also**

TRC\_disable  
TRC\_query  
LOG\_printf  
LOG\_event  
STS\_add  
STS\_delta

TRC\_query

Query trace class(es)

C Interface

Syntax	result = TRC_query(mask);		
Parameters	Uns	mask;	/* trace type constant mask */
Return Value	Int	result	/* indicates whether all trace types enabled */

Assembly Interface



Syntax	TRC_query mask
Inputs	mask
Preconditions	constant - mask for trace types
Postconditions	a == 0 if all trace types in the mask are enabled a != 0 if any trace type in the mask is disabled
Modifies	ag, ah, al, c

Assembly Interface



Syntax	TRC_query mask
Inputs	mask
Preconditions	constant - mask for trace types
Postconditions	t0 == 0 if all trace types in the mask are enabled t0 != 0 if any trace type in the mask is disabled
Modifies	t0

Reentrant                    yes

Description                TRC\_query determines whether particular trace types are enabled. TRC\_query returns 0 if all trace types in the mask are enabled. If any trace types in the mask are disabled, TRC\_query returns a value with a

bit set for each trace type in the mask that is disabled. (See the TRC Module for a list of constants to use in the mask.)

Trace types are specified with a 16-bit mask. The full list of constants you can use is included in the description of the TRC module.

For example, the following C code returns 0 if statistics tracing for the PRD class is enabled:

```
result = TRC_query(TRC_STSPRD);
```

The following C code returns 0 if both logging and statistics tracing for the SWI class are enabled:

```
result = TRC_query(TRC_LOGSWI | TRC_STSSWI);
```

Note that TRC\_query does not return 0 unless the bits you are querying and the TRC\_GBLHOST and TRC\_GBLTARG bits are set. TRC\_query returns non-zero if either TRC\_GBLHOST or TRC\_GBLTARG are disabled. This is because no tracing is done unless these bits are set.

For example, if the TRC\_GBLHOST, TRC\_GBLTARG, and TRC\_LOGSWI bits are set, the following C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)      /* returns 0 */
result = TRC_query(TRC_LOGPRD)      /* returns non-zero */
*/
```

However, if only the TRC\_GBLHOST and TRC\_LOGSWI bits are set, the same C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)      /* returns non-zero */
*/
result = TRC_query(TRC_LOGPRD)      /* returns non-zero */
*/
```

## See Also

TRC\_enable  
TRC\_disable

## 2.24 TSK Module

The TSK module is the task manager.*r*

### Functions

- ☐ TSK\_checkstacks. Check for stack overflow
- ☐ TSK\_create. Create a task ready for execution
- ☐ TSK\_delete. Delete a task
- ☐ TSK\_deltatime. Update task STS with time difference
- ☐ TSK\_disable. Disable DSP/BIOS task scheduler
- ☐ TSK\_enable. Enable DSP/BIOS task scheduler
- ☐ TSK\_exit. Terminate execution of the current task
- ☐ TSK\_getenv. Get task environment
- ☐ TSK\_geterr. Get task error number
- ☐ TSK\_getname. Get task name
- ☐ TSK\_getpri. Get task priority
- ☐ TSK\_getsts. Get task STS object
- ☐ TSK\_itick. Advance system alarm clock (interrupt only)
- ☐ TSK\_self. Get handle of currently executing task
- ☐ TSK\_setenv. Set task environment
- ☐ TSK\_seterr. Set task error number
- ☐ TSK\_setpri. Set a task's execution priority
- ☐ TSK\_settime. Set task STS previous time
- ☐ TSK\_sleep. Delay execution of the current task
- ☐ TSK\_stat. Retrieve the status of a task
- ☐ TSK\_tick. Advance system alarm clock
- ☐ TSK\_time. Return current value of system clock
- ☐ TSK\_yield. Yield processor to equal priority task

### Task Hook Functions

```
Void TSK_createFxn(TSK_Handle task);
Void TSK_deleteFxn(TSK_Handle task);
Void TSK_exitFxn(TSK_Handle task);
Void TSK_readyFxn(TSK_Handle newtask);
Void TSK_switchFxn(KNL_Handle oldtask,
                  KNL_Handle newtask);
```



## Constants, Types, and Structures

```

typedef struct TSK_OBJ *TSK_Handle;
/* handle for task object */

struct TSK_Attrs { /* task attributes */
    Int    priority; /* execution priority */
    Ptr    stack;    /* pre-allocated stack */
    Uns    stacksize; /* stack size in MADUs */
#ifdef _55_
    Uns    sysstacksize; /*C55x system stack in MADUs */
#endif
    Int    stackseg; /* memory seg for stack allocation */
    Ptr    environ; /* global environment data structure */
    String name;    /* printable name */
    Bool    exitflag; /* program termination requires this */
/* task to terminate */
    TSK_DBG_Mode debug /* indicates enum type TSK_DBG_YES */
/* TSK_DBG_NO or TSK_DBG_MAYBE */
};

Int TSK_pid; /* MP processor ID */

Int TSK_MAXARGS = 8; /* maximum number of task arguments */
Int TSK_IDLEPRI = 0; /* used for idle task */
Int TSK_MINPRI = 1; /* minimum execution priority */
Int TSK_MAXPRI = 15; /* maximum execution priority */
Int TRG_STACKSTAMP =
TSK_Attrs TSK_ATTRS = { /* default attribute values */
    TSK->PRIORITY, /* priority */
    NULL, /* stack */
    TSK->STACKSIZE, /* stacksize */
#ifdef _55_
    TSK->SYSSTACKSIZE, /* C55x system stacksize in MADUs */
#endif
    TSK->STACKSEG, /* stackseg */
    NULL, /* environ */
    "", /* name */
    TRUE, /* exitflag */
};

enum TSK_Mode { /* task execution modes */
    TSK_RUNNING, /* task is currently executing */
    TSK_READY, /* task is scheduled for execution */
    TSK_BLOCKED, /* task is suspended from execution */
    TSK_TERMINATED, /* task is terminated from execution */
};

struct TSK_Stat { /* task status structure */
    TSK_Attrs attrs; /* task attributes */
    TSK_Mode mode; /* task execution mode */
    Ptr sp; /* task stack pointer */
    Uns used; /* task stack used */
};

```

## Description

The TSK module makes available a set of functions that manipulate task objects accessed through handles of type `TSK_Handle`. Tasks represent independent threads of control that conceptually execute functions in parallel within a single C program; in reality, concurrency is achieved by switching the processor from one task to the next.

When you create each task, it is provided with its own run-time stack, used for storing local variables as well as for further nesting of function calls. The `TRG_STACKSTAMP` value is used to initialize the run-time stack. 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. 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.

Each task is in one of four modes of execution at any point in time: running, ready, blocked, or terminated. By design, there is always one (and only one) task currently running, even if it is a dummy idle task managed internally by TSK. The current task can be suspended from execution by calling certain TSK functions, as well as functions provided by other modules like SEM and SIO; the current task can also terminate its execution. In either case, the processor is switched to the next task that is ready to run.

You can assign numeric priorities to tasks through TSK. Tasks are readied for execution in strict priority order; tasks of the same priority are scheduled on a first-come, first-served basis. As a rule, the priority of the currently running task is never lower than the priority of any ready task. Conversely, the running task is preempted and re-scheduled for execution whenever there exists some ready task of higher priority.

You can use the DSP/BIOS Configuration Tool to specify application-wide task hook functions that run whenever a task state changes in a particular way. These functions are the Create, Delete, Exit, Switch, and Ready functions. The `TSK_create` topic describes the Create function. The `TSK_delete` topic describes the Delete function. The `TSK_exit` topic describes the Exit function.

If a Switch function is specified, it is invoked when a new task becomes the `TSK_RUNNING` task. The Switch function gives the application access to both the current and next task handles at task switch time. The function should use these argument types:

```
void mySwitchFxn(TSK_Handle currTask,  
                TSK_Handle nextTask);
```

This function can be used to save/restore additional task context (for example, external hardware registers), to check for task stack overflow, to monitor the time used by each task, etc.

If a Ready function is specified, it is invoked whenever a task is made ready to run. Even if a higher-priority thread is running, the Ready function runs. The Ready function will be called with a handle to the task being made ready to run as its argument. This example function prints the name of both the task that is ready to run and the task that is currently running:

```
void myReadyFxn(TSK_Handle task)
{
    String      nextName, currName;
    TSK_Handle  currTask = TSK_self();

    nextName = TSK_getname(task);
    LOG_printf(&trace, "Task %s Ready", nextName);

    currName = TSK_getname(currTask);
    LOG_printf(&trace, "Task %s Running", currName);
}
```

The Switch function and Ready function are called in such a way that they can use only functions allowed within a software interrupt handler. See Appendix A, Function Callability and Error Tables, for a list of functions that can be called by SWI handlers. There are no real constraints on what functions are called via the Create function, Delete function, or Exit function.

## TSK Manager Properties

The following global properties can be set for the TSK module on the TSK Manager Properties dialog in the Configuration Tool:

- ☐ **Enable TSK Manager.** If no tasks are used by the program other than `TSK_idle`, you can optimize the program by disabling the task manager. The program must then not use TSK objects created with either the Configuration Tool or the `TSK_create` function. If the task manager is disabled, the idle loop still runs and uses the system stack instead of a task stack.
- ☐ **Object Memory.** The memory segment that contains the TSK objects created with the Configuration Tool.
- ☐ **Default stack size.** The default size of the stack (in MADUs) used by tasks. You can override this value for an individual task you create with the Configuration Tool or `TSK_create`. The estimated minimum task size is shown in the status bar of the Configuration Tool. This property applies to TSK objects created both with the Configuration Tool and with `TSK_create`.



- ☐ **Default systack size.** This property defines the size (in MADUs) of the system stack.
- ☐ **Stack segment for dynamic tasks.** The default memory segment that will contain task objects created at run-time with the TSK\_create function. The TSK\_Attrs structure passed to the TSK\_create function can override this default. If you select MEM\_NULL for this property, creation of task objects at run-time is disabled.
- ☐ **Default task priority.** The default priority level for tasks that are created dynamically with TSK\_create. This property applies to TSK objects created both with the Configuration Tool and with TSK\_create.
- ☐ **TSK tick driven by.** Choose whether you want the system clock to be driven by the PRD module or by calls to TSK\_tick and TSK\_itick. This clock is used by TSK\_sleep and functions such as SEM\_pend that accept a timeout argument.
- ☐ **Create function.** The name of a function to call when any task is created at run-time with TSK\_create. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.) The TSK\_create topic describes the Create function.
- ☐ **Delete function.** The name of a function to call when any task is deleted at run-time with TSK\_delete. If this function is written in C, use a leading underscore before the C function name. The TSK\_delete topic describes the Delete function.
- ☐ **Exit function.** The name of a function to call when any task exits. If this function is written in C, use a leading underscore before the C function name. The TSK\_exit topic describes the Exit function.
- ☐ **Call switch function.** Check this box if you want a function to be called when any task switch occurs.
- ☐ **Switch function.** The name of a function to call when any task switch occurs. This function can give the application access to both the current and next task handles. If this function is written in C, use a leading underscore before the C function name. The TSK Module topic describes the Switch function.
- ☐ **Call ready function.** Check this box if you want a function to be called when any task becomes ready to run.
- ☐ **Ready function.** The name of a function to call when any task becomes ready to run. If this function is written in C, use a leading underscore before the C function name. The TSK Module topic describes the Ready function.

## TSK Object Properties

The following properties can be set for a TSK object on the TSK Object Properties dialog in the Configuration Tool:

### General tab

- ☐ **comment.** A comment to identify this TSK object.
- ☐ **Automatically allocate stack.** Check this box if you want the task's private stack space to be allocated automatically when this task is created. The task's context is saved in this stack before any higher-priority task is allowed to block this task and run.
- ☐ **Manually allocated stack.** If you did not check the box to Automatically allocate stack, type the name of the manually allocated stack to use for this task. If the stack is defined in a C program, add a leading underscore before the stack name.
- ☐ **Stack size.** If you checked the box to Automatically allocate stack, enter the size (in MADUs) of the stack space to allocate for this task. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context plus the maximum nested interrupt 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.
- ☐ **System stack size.** This specifies the size (in MADUs) of the task's system stack
- ☐ **Stack Memory Segment.** If you checked the box to Automatically allocate stack, select the memory segment to contain the stack space for this task.
- ☐ **Priority.** The priority level for this task.



### Function tab

- ☐ **Task function.** The function to be executed when the task runs. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- ☐ **Task function argument 0-7.** The arguments to pass to the task function. Arguments can be integers or labels. For labels defined in a C program, add a leading underscore before the label name.

### Advanced tab

- ☐ **Environment pointer.** A pointer to a globally defined data structure that this task can access. The task can get and set the task environment pointer with the `TSK_getenv` and `TSK_setenv` functions. If this structure is defined in C, use a leading underscore before the structure name.
- ☐ **Don't shut down system while this task is still running.** Check this box if you do not want the application to be able to end if this task is still running. The application can still abort. For example, you might

clear this box for a monitor task that collects data whenever all other tasks are blocked. The application does not need to explicitly shut down this task.

- ☐ **Allocate Task Name on Target.** Check this box if you want the name of this TSK object to be retrievable by the TSK\_getname function. Clearing this box saves a small amount of memory. The task name is available in analysis tools in either case.

## **TSK - DSP/BIOS Analysis Tool Interface**

The TSK tab of the Kernel/Object View shows information about task objects.

To enable TSK logging, choosing DSP/BIOS→RTA Control Panel and check the appropriate box. Then you can open the system log by choosing View→System Log. You see a graph of activity that includes TSK function execution states.

Only TSK objects created with the Configuration Tool are traced. The System Log graph includes time spent performing dynamically created TSK functions in the Other Threads row.

You can also enable TSK accumulators in the RTA Control Panel. Then you can choose DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a TSK object, you see statistics about the time elapsed from the time the TSK was posted (made ready to run) until TSK\_deltatime is called by the application. See TSK\_settime on page 2-349 and TSK\_deltatime on page 2-334, for more information on gathering statistics on TSK objects.

**TSK\_checkstacks** *Check for stack overflow***C Interface**

<b>Syntax</b>	<code>TSK_checkstacks(oldtask, newtask);</code>
<b>Parameters</b>	<code>TSK_Handle oldtask; /* handle of task switched from */</code> <code>TSK_Handle newtask; /* handle of task switched to */</code>
<b>Return Value</b>	Void

**Assembly Interface** none

**Description** TSK\_checkstacks calls SYS\_abort with an error message if either oldtask or newtask has a stack in which the last location no longer contains the initial value TRG\_STACKSTAMP. The presumption in one case is that oldtask's stack overflowed, and in the other that an invalid store has corrupted newtask's stack.

You can call TSK\_checkstacks directly from your application. For example, you can check the current task's stack integrity at any time with a call like the following:

```
TSK_checkstacks(TSK_self(), TSK_self());
```

However, it is more typical to call TSK\_checkstacks in the task Switch function specified for the TSK manager in your configuration file. This provides stack checking at every context switch, with no alterations to your source code.

If you want to perform other operations in the Switch function, you can do so by writing your own function (myswitchfxn) and then calling TSK\_checkstacks from it.

```
Void myswitchfxn(TSK_Handle oldtask,  
                 TSK_Handle newtask)  
{  
    `your additional context switch operations`  
    TSK_checkstacks(oldtask, newtask);  
    ...  
}
```

**Constraints and  
Calling Context**

- ❑ TSK\_checkstacks cannot be called from an HWI or SWI.

**TSK\_create***Create a task ready for execution***C Interface****Syntax**

```
task = TSK_create(fxn, attrs, [arg,] ...);
```

**Parameters**

```
Fxn      fxn;      /* entry point of the task */
TSK_Attrs *attrs;   /* pointer to task attributes */
Arg      arg;      /* task arguments */
```

**Return Value**

```
TSK_Handle task;    /* task object handle */
```

**Assembly Interface**

```
none
```

**Description**

TSK\_create creates a new task object. If successful, TSK\_create returns the handle of the new task object. If unsuccessful, TSK\_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort).

You can use the DSP/BIOS Configuration Tool to specify an application-wide Create function that runs whenever a task is created. The default Create function is a no-op function. The Create function is called after the task handle has been initialized but before the task has been placed on its ready queue. Any DSP/BIOS function can be called from the Create function. DSP/BIOS passes the task handle of the task being created to your Create function. Your Create function declaration should be similar to the following:

```
Void myCreateFxn(TSK_Handle task);
```

The new task is placed in TSK\_READY mode, and is scheduled to begin concurrent execution of the following function call:

```
(*fxn)(arg1, arg2, ... argN) /* N = TSK_MAXARGS = 8 */
```

As a result of being made ready to run, the task runs the application-wide Ready function if one has been specified.

TSK\_exit is automatically called if and when the task returns from fxn.

If attrs is NULL, the new task is assigned a default set of attributes. Otherwise, the task's attributes are specified through a structure of type TSK\_Attrs defined as follows:



```
struct TSK_Attrs {
    Int      priority;
    Ptr      stack;
    Uns      stacksize;
#ifdef _55_
    /*C55x system stack size in MADUs*/
    Uns sysstacksize;
#endif
    Uns      stackseg;
    Ptr      environ;
    String   name;
    Bool     exitflag;
};
```

The priority attribute specifies the task's execution priority and must be less than or equal to TSK\_MAXPRI (15); this attribute defaults to the value of the configuration parameter Default task priority (preset to TSK\_MINPRI). If priority is less than 0, task is barred from execution until its priority is raised at a later time by another task. A priority value of 0 is reserved for the TSK\_idle task defined in the default configuration. You should not use a priority of 0 for any other tasks.

The stack attribute specifies a pre-allocated block of stacksize MADUs to be used for the task's private stack; this attribute defaults to NULL, in which case the task's stack is automatically allocated using MEM\_alloc from the memory segment given by the stackseg attribute.

The stacksize attribute specifies the number of MADUs to be allocated for the task's private stack; this attribute defaults to the value of the configuration parameter Default stack size (preset to 1024). Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context plus the maximum nested interrupt 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.

The stackseg attribute specifies the memory segment to use when allocating the task stack with MEM\_alloc; this attribute defaults to the value of the configuration parameter Default stack segment.



The system stack attribute specifies a pre-allocated block of sysstacksize MADUs to be used for the task's private system stack. This attribute defaults to NULL, in which case the task's system stack is automatically allocated using MEM\_alloc from the memory segment given by the stackseg attribute. The sysstacksize attribute specifies the number of MADUs to be allocated for the task's private system stack. This attribute defaults to the value of the configuration parameter Default system stack size (preset to 256).

The `environ` attribute specifies the task's global environment through a generic pointer that references an arbitrary application-defined data structure; this attribute defaults to `NULL`.

The `name` attribute specifies the task's printable name, which is a `NULL`-terminated character string; this attribute defaults to the empty string `""`. This name can be returned by `TSK_getname`.

The `exitflag` attribute specifies whether or not the task must terminate before the program as a whole can terminate; this attribute defaults to `TRUE`.

All default attribute values are contained in the constant `TSK_ATTRS`, which can be assigned to a variable of type `TSK_Attrs` prior to calling `TSK_create`.

A task switch occurs when calling `TSK_create` if the priority of the new task is greater than the priority of the current task.

`TSK_create` calls `MEM_alloc` to dynamically create the object's data structure. `MEM_alloc` must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM module, page 2–136.

## Constraints and Calling Context

- ❑ `TSK_create` cannot be called from an SWI or HWI.
- ❑ The `fxn` parameter and the `name` attribute cannot be `NULL`.
- ❑ The `priority` attribute must be less than or equal to `TSK_MAXPRI` and greater than or equal to `TSK_MINPRI`. The `priority` can be less than zero (0) for tasks that should not execute.
- ❑ The string referenced through the `name` attribute cannot be allocated locally.
- ❑ The `stackseg` attribute must identify a valid memory segment.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the `XXX_create` functions.

## See Also

`MEM_alloc`  
`SYS_error`  
`TSK_delete`  
`TSK_exit`

TSK\_delete

Delete a task

C Interface

Syntax	TSK_delete(task);
Parameters	TSK_Handle task;      /* task object handle */
Return Value	Void

Assembly Interface      none

**Description**      TSK\_delete removes the task from all internal queues and calls MEM\_free to free the task object and stack. task should be in a state that does not violate any of the listed constraints.

If all remaining tasks have their exitflag attribute set to FALSE, DSP/BIOS terminates the program as a whole by calling SYS\_exit with a status code of 0.

You can use the DSP/BIOS Configuration Tool to specify an application-wide Delete function that runs whenever a task is deleted. The default Delete function is a no-op function. The Delete function is called before the task object has been removed from any internal queues and its object and stack are freed. Any DSP/BIOS function can be called from the Delete function. DSP/BIOS passes the task handle of the task being deleted to your Delete function. Your Delete function declaration should be similar to the following:

```
Void myDeleteFxn(TSK_Handle task);
```

TSK\_delete calls MEM\_free to delete the TSK object. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

---

**Note:**

Unless the mode of the deleted task is TSK\_TERMINATED, TSK\_delete should be called with care. For example, if the task has obtained exclusive access to a resource, deleting the task makes the resource unavailable.

---

Constraints and  
Calling Context

- ❑ The task cannot be the currently executing task (TSK\_self).
- ❑ TSK\_delete cannot be called from an SWI or HWI.

- ❑ No check is performed to prevent TSK\_delete from being used on a statically-created object. If a program attempts to delete a task object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

MEM\_free  
TSK\_create

**TSK\_deltatime**

*Update task statistics with difference between current time and time task was made ready*

**C Interface**

<b>Syntax</b>	TSK_deltatime(task);
<b>Parameters</b>	TSK_Handle task;     /* task object handle */
<b>Return Value</b>	Void

**Assembly Interface**     none

**Description**

This function accumulates the time difference from when a task is made ready to the time TSK\_deltatime is called. These time differences are accumulated in the task's internal STS object and can be used to determine whether or not a task misses real-time deadlines.

If TSK\_deltatime is not called by a task, its STS object will never be updated in the Statistics View, even if TSK accumulators are enabled in the RTA Control Panel.

TSK statistics are handled differently than other statistics because TSK functions typically run an infinite loop that blocks when waiting for other threads. In contrast, HWI and SWI functions run to completion without blocking. Because of this difference, DSP/BIOS allows programs to identify the "beginning" of a TSK function's processing loop by calling TSK\_settime and the "end" of the loop by calling TSK\_deltatime.

For example, if a task waits for data and then processes the data, you want to ensure that the time from when the data is made available until the processing is complete is always less than a certain value. A loop within the task can look something like the following:

```
Void task
{
    'do some startup work'

    /* Initialize time in task's
       STS object to current time */
    TSK_settime(TSK_self);

    for (;;) {
        /* Get data */
        SIO_get(...);

        'process data'
```

```
/* Get time difference and  
   add it to task's STS object */  
TSK_deltatime(TSK_self);  
}
```

In the example above, the task blocks on SIO\_get and the device driver posts a semaphore that readies the task. DSP/BIOS sets the task's statistics object with the current time when the semaphore becomes available and the task is made ready to run. Thus, the call to TSK\_deltatime effectively measures the processing time of the task.

### **Constraints and Calling Context**

- ❑ The results of calls to TSK\_deltatime and TSK\_settime are displayed in the Statistics View only if Enable TSK accumulators is selected in the RTA Control Panel.

### **See Also**

TSK\_getsts  
TSK\_settime

**TSK\_disable***Disable DSP/BIOS task scheduler***C Interface****Syntax** TSK\_disable();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_disable disables the DSP/BIOS task scheduler. The current task continues to execute (even if a higher priority task can become ready to run) until TSK\_enable is called.

TSK\_disable does not disable interrupts, but is instead used before disabling interrupts to make sure a context switch to another task does not occur when interrupts are disabled.

TSK\_disable handlers are disabled. TSK\_disable maintains a count which allows nested calls to TSK\_disable. Task switching is not reenabled until TSK\_enable has been called as many times as TSK\_disable. Calls to TSK\_disable can be nested.

Since TSK\_disable can prohibit ready tasks of higher priority from running it should not be used as a general means of mutual exclusion. SEM semaphores should be used for mutual exclusion when possible.

**Constraints and  
Calling Context**

- ☐ No kernel operations that can cause the current task to block can be made from within a TSK\_disable / TSK\_enable block. This includes SEM\_pend (unless timeout is 0), TSK\_sleep, and TSK\_yield.
- ☐ TSK\_yield cannot be called within a TSK\_disable / TSK\_enable block.
- ☐ TSK\_disable cannot be called from an SWI or HWI.
- ☐ TSK\_disable cannot be called from the program's main function.

**See Also**

SEM Module  
TSK\_enable

**TSK\_enable***Enable DSP/BIOS task scheduler***C Interface****Syntax** TSK\_enable();**Parameters** Void**Return Value** Void**Assembly Interface** none

**Description** TSK\_enable is used to reenable the DSP/BIOS task scheduler after TSK\_disable has been called. Since TSK\_disable calls can be nested, the task scheduler is not enabled until TSK\_enable is called the same number of times as TSK\_disable.

A task switch occurs when calling TSK\_enable only if there exists a TSK\_READY task whose priority is greater than the currently executing task.

**Constraints and Calling Context**

- ❑ No kernel operations that can cause the current task to block (for example, SEM\_pend, TSK\_sleep, TSK\_yield) can be made from within a TSK\_disable / TSK\_enable block.
- ❑ TSK\_enable cannot be called from an SWI or HWI.
- ❑ TSK\_enable cannot be called from the program's main function.

**See Also** SEM Module  
TSK\_disable



**TSK\_exit***Terminate execution of the current task***C Interface****Syntax** TSK\_exit();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_exit terminates execution of the current task, changing its mode from TSK\_RUNNING to TSK\_TERMINATED. If all tasks have been terminated, or if all remaining tasks have their exitflag attribute set to FALSE, then DSP/BIOS terminates the program as a whole by calling the function SYS\_exit with a status code of 0.

TSK\_exit is automatically called whenever a task returns from its top-level function.

You can use the DSP/BIOS Configuration Tool to specify an application-wide Exit function that runs whenever a task is terminated. The default Exit function is a no-op function. The Exit function is called before the task has been blocked and marked TSK\_TERMINATED. Any DSP/BIOS function can be called from the Exit function. DSP/BIOS passes the task handle of the task being exited to your Exit function. Your Exit function declaration should be similar to the following:

```
Void myExitFxn(TSK_Handle task);
```

A task switch occurs when calling TSK\_exit unless the program as a whole is terminated.

**Constraints and Calling Context**

- ☐ TSK\_exit cannot be called from an SWI or HWI.
- ☐ TSK\_exit cannot be called from the program's main function.

**See Also**

MEM\_free  
TSK\_create  
TSK\_delete

**TSK\_getenv***Get task environment pointer***C Interface****Syntax**`environ = TSK_getenv(task);`**Parameters**`TSK_Handle task;      /* task object handle */`**Return Value**`Ptr                  environ;   /* task environment pointer */`**Assembly Interface**

none

**Description**

TSK\_getenv returns the environment pointer of task. The environment pointer, environ, references an arbitrary application-defined data structure.

**See Also**

TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_geterr***Get task error number***C Interface**

<b>Syntax</b>	<code>errno = TSK_geterr(task);</code>
<b>Parameters</b>	<code>TSK_Handle task;</code> <i>/* task object handle */</i>
<b>Return Value</b>	<code>Int            errno;</code> <i>/* error number */</i>

**Assembly Interface**      none

**Description**      Each task carries a task-specific error number. This number is initially SYS\_OK, but it can be changed by TSK\_seterr. TSK\_geterr returns the current value of this number.

**See Also**      SYS\_error  
TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_getname***Get task name***C Interface**

**Syntax**                      `name = TSK_getname(task);`

**Parameters**                `TSK_Handle task;      /* task object handle */`

**Return Value**              `String            name;      /* task name */`

**Assembly Interface**

none

**Description**

TSK\_getname returns the task's name.

For tasks created with the Configuration Tool, the name is available to this function only if the Allocate Task Name on Target box is checked in the properties for this task. For tasks created with TSK\_create, TSK\_getname returns the `attrs.name` field value, or an empty string if this attribute was not specified.

**See Also**

TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_getpri***Get task priority***C Interface**

<b>Syntax</b>	priority = TSK_getpri(task);
<b>Parameters</b>	TSK_Handle task;     /* task object handle */
<b>Return Value</b>	Int                    priority;   /* task priority */

**Assembly Interface**                none

**Description**                        TSK\_getpri returns the priority of task.

**See Also**                            TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_getsts***Get the handle of the task's STS object***C Interface**

<b>Syntax</b>	<code>sts = TSK_getsts(task);</code>
<b>Parameters</b>	<code>TSK_Handle task;</code> <i>/* task object handle */</i>
<b>Return Value</b>	<code>STS_Handle sts;</code> <i>/* statistics object handle */</i>

**Assembly Interface**      none

**Description**      This function provides access to the task's internal STS object. For example, you can want the program to check the maximum value to see if it has exceeded some value.

**See Also**      TSK\_deltatime  
TSK\_settime

**TSK\_itick***Advance the system alarm clock (interrupt use only)***C Interface****Syntax** TSK\_itick();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description** TSK\_itick increments the system alarm clock, and readies any tasks blocked on TSK\_sleep or SEM\_pend whose timeout intervals have expired.**Constraints and  
Calling Context**

- ☐ TSK\_itick cannot be called by a TSK object.
- ☐ TSK\_itick cannot be called from the program's main function.
- ☐ When called within an HWI ISR, the code sequence calling TSK\_itick must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**See Also**SEM\_pend  
TSK\_sleep  
TSK\_tick

**TSK\_self***Returns handle to the currently executing task***C Interface****Syntax**`curtask = TSK_self();`**Parameters**

Void

**Return Value**

TSK\_Handle curtask; /\* handle for current task object \*/

**Assembly Interface**

none

**Description**

TSK\_self returns the object handle for the currently executing task. This function is useful when inspecting the object or when the current task changes its own priority through TSK\_setpri.

No task switch occurs when calling TSK\_self.

**See Also**

TSK\_setpri



**TSK\_setenv**

*Set task environment*

**C Interface**

<b>Syntax</b>	TSK_setenv(task, environ);
<b>Parameters</b>	TSK_Handle task;     /* task object handle */ Ptr            environ; /* task environment pointer */
<b>Return Value</b>	Void
<b>Assembly Interface</b>	none
<b>Description</b>	TSK_setenv sets the task environment pointer to environ. The environment pointer, environ, references an arbitrary application-defined data structure.
<b>See Also</b>	TSK_getenv TSK_geterr

**TSK\_seterr***Set task error number***C Interface**

**Syntax** TSK\_seterr(task, errno);

**Parameters** TSK\_Handle task; /\* task object handle \*/  
Int errno; /\* error number \*/

**Return Value** Void

**Assembly Interface** none

**Description** Each task carries a task-specific error number. This number is initially SYS\_OK, but can be changed to errno by calling TSK\_seterr. TSK\_geterr returns the current value of this number.

**See Also** TSK\_getenv  
TSK\_geterr

**TSK\_setpri***Set a task's execution priority***C Interface**

<b>Syntax</b>	<code>oldpri = TSK_setpri(task, newpri);</code>
<b>Parameters</b>	<code>TSK_Handle task;</code> /* task object handle */ <code>Int newpri;</code> /* task's new priority */
<b>Return Value</b>	<code>Int oldpri;</code> /* task's old priority */

**Assembly Interface**

none

**Description**

TSK\_setpri sets the execution priority of task to newpri, and returns that task's old priority value. Raising or lowering a task's priority does not necessarily force preemption and re-scheduling of the caller: tasks in the TSK\_BLOCKED mode remain suspended despite a change in priority; and tasks in the TSK\_READY mode gain control only if their (new) priority is greater than that of the currently executing task.

The maximum value of newpri is TSK\_MAXPRI(15). If the minimum value of newpri is TSK\_MINPRI(0). If newpri is less than 0, task is barred from further execution until its priority is raised at a later time by another task; if newpri equals TSK\_MAXPRI, execution of task effectively locks out all other program activity, except for the handling of interrupts.

The current task can change its own priority (and possibly preempt its execution) by passing the output of TSK\_self as the value of the task parameter.

A context switch occurs when calling TSK\_setpri if a task makes its own priority lower than the priority of another currently ready task, or if the currently executing task makes a ready task's priority higher than its own priority. TSK\_setpri can be used for mutual exclusion.

**Constraints and Calling Context**

- ☐ newpri must be less than or equal to TSK\_MAXPRI.
- ☐ The task cannot be TSK\_TERMINATED.
- ☐ The new priority should not be zero (0). This priority level is reserved for the TSK\_idle task.

**See Also**

TSK\_self  
TSK\_sleep

**TSK\_settime***Reset task statistics previous value to current time***C Interface**

<b>Syntax</b>	TSK_settime(task);
<b>Parameters</b>	TSK_Handle task;     /* task object handle */
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

Your application can call TSK\_settime before a task enters its processing loop in order to ensure your first call to TSK\_deltatime is as accurate as possible and doesn't reflect the time difference since the time the task was created. However, it is only necessary to call TSK\_settime once for initialization purposes. After initialization, DSP/BIOS sets the time value of the task's STS object every time the task is made ready to run.

TSK statistics are handled differently than other statistics because TSK functions typically run an infinite loop that blocks when waiting for other threads. In contrast, HWI and SWI functions run to completion without blocking. Because of this difference, DSP/BIOS allows programs to identify the "beginning" of a TSK function's processing loop by calling TSK\_settime and the "end" of the loop by calling TSK\_deltatime.

For example, a loop within the task can look something like the following:

```
Void task
{
    'do some startup work'

    /* Initialize time in task's
       STS object to current time */
    TSK_settime(TSK_self());

    for (;;) {
        /* Get data */
        SIO_get(...);

        'process data'

        /* Get time difference and
           add it to task's STS object */
        TSK_deltatime(TSK_self());
    }
}
```

In the previous example, the task blocks on SIO\_get and the device driver posts a semaphore that readies the task. DSP/BIOS sets the task's statistics object with the current time when the semaphore becomes available and the task is made ready to run. Thus, the call to TSK\_deltatime effectively measures the processing time of the task.

**Constraints and  
Calling Context**

- ❑ TSK\_settime cannot be called from the program's main function.
- ❑ The results of calls to TSK\_deltatime and TSK\_settime are displayed in the Statistics View only if Enable TSK accumulators is selected within the RTA Control Panel.

**See Also**

TSK\_deltatime  
TSK\_getsts

**TSK\_sleep***Delay execution of the current task***C Interface**

<b>Syntax</b>	TSK_sleep(nticks);
<b>Parameters</b>	Uns            nticks;    /* number of system clock ticks to sleep */
<b>Return Value</b>	Void

**Assembly Interface**

none

**Description**

TSK\_sleep changes the current task's mode from TSK\_RUNNING to TSK\_BLOCKED, and delays its execution for nticks increments of the system clock. The actual time delayed can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

After the specified period of time has elapsed, the task reverts to the TSK\_READY mode and is scheduled for execution.

A task switch always occurs when calling TSK\_sleep if nticks > 0.

**Constraints and Calling Context**

- ☐ TSK\_sleep cannot be called from an SWI or HWI, or within a TSK\_disable / TSK\_enable block.
- ☐ TSK\_sleep cannot be called from the program's main function.
- ☐ TSK\_sleep should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.
- ☐ nticks cannot be SYS\_FOREVER.

**TSK\_stat***Retrieve the status of a task***C Interface****Syntax**

TSK\_stat(task, statbuf);

**Parameters**

TSK\_Handle task;     /\* task object handle \*/  
TSK\_Stat     \*statbuf; /\* pointer to task status structure \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

TSK\_stat retrieves attribute values and status information about task; the current task can inquire about itself by passing the output of TSK\_self as the first argument to TSK\_stat.

Status information is returned through statbuf, which references a structure of type TSK\_Stat defined as follows:

```
struct TSK_Stat {     /* task status structure */  
    TSK_Attrs attrs; /* task attributes */  
    TSK_Mode mode; /* task execution mode */  
    Ptr     sp;     /* task's current stack pointer */  
    Uns     used; /* max number of words ever */  
                 /* used on the task stack */  
};
```

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.

TSK\_stat has a non-deterministic execution time. As such, it is not recommended to call this API from SWIs or HWIs.

**Constraints and  
Calling Context**

❑ statbuf cannot be NULL.

**See Also**

TSK\_create

**TSK\_tick***Advance the system alarm clock***C Interface****Syntax** TSK\_tick();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_tick increments the system clock, and readies any tasks blocked on TSK\_sleep or SEM\_pend whose timeout intervals have expired. TSK\_tick can be invoked by an ISR or by the currently executing task. The latter is particularly useful for testing timeouts in a controlled environment.

A task switch occurs when calling TSK\_tick if the priority of any of the readied tasks is greater than the priority of the currently executing task.

**Constraints and  
Calling Context**

- ❑ When called within an HWI ISR, the code sequence calling TSK\_tick must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.

**See Also**

CLK Module  
SEM\_pend  
TSK\_itick  
TSK\_sleep



**TSK\_time***Return current value of system clock***C Interface**

<b>Syntax</b>	<code>curtime = TSK_time();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	Uns <code>curtime; /* current time */</code>

**Assembly Interface**

none

**Description**

TSK\_time returns the current value of the system alarm clock.

Note that since the system clock is usually updated asynchronously by an interrupt service routine (via TSK\_itick or TSK\_tick), curtime can lag behind the actual system time. This lag can be even greater if a higher priority task preempts the current task between the call to TSK\_time and when its return value is used. Nevertheless, TSK\_time is useful for getting a rough idea of the current system time.

**TSK\_yield***Yield processor to equal priority task***C Interface****Syntax** TSK\_yield();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description** TSK\_yield yields the processor to another task of equal priority.

A task switch occurs when you call TSK\_yield if there is an equal priority task ready to run.

**Constraints and Calling Context**

- ❑ When called within an HWI ISR, the code sequence calling TSK\_yield must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.
- ❑ TSK\_yield cannot be called from the program's main function.
- ❑ TSK\_yield should not be called from within a TSK\_disable/TSK\_enable block.

**See Also** TSK\_sleep

## 2.25 std.h and stdlib.h functions

This section contains descriptions of special utility macros found in std.h and DSP/BIOS standard library functions found in stdlib.h .

### Macros

- ❑ **ArgToInt.** Cast an Arg type parameter as an integer type.
- ❑ **ArgToPtr.** Cast an Arg type parameter as a pointer type.

### Functions

- ❑ **atexit.** Register an exit function.
- ❑ **\*calloc.** Allocate and clear memory.
- ❑ **exit.** Call the exit functions registered by atexit.
- ❑ **free.\*getenv.** Get environmental variable.
- ❑ **\*malloc.** Allocate memory.
- ❑ **\*realloc.** Reallocate a memory packet.

### Syntax

```
#include <std.h>
ArgToInt(arg)
ArgToPtr(arg)
```

```
#include <stdlib.h>
int atexit(void (*fcn)(void));
void *calloc(size_t nobj, size_t size);
void exit(int status);
void free(void *p);
char *getenv(char *name);
void *malloc(size_t size);
void *realloc(void *p, size_t size);
```

### Description

The DSP/BIOS library contains some C standard library functions which supersede the library functions bundled with the C compiler. These functions follow the ANSI C specification for parameters and return values. Consult Kernighan and Ritchie for a complete description of these functions.

The functions calloc, free, malloc, and realloc use MEM\_alloc and MEM\_free (with segid = Segment for malloc/free) to allocate and free memory.

getenv uses the \_environ variable defined and initialized in the boot file to search for a matching environment string.

exit calls the exit functions registered by atexit before calling SYS\_exit.

# Utility Programs

---

---

---

This chapter provides documentation for TMS320C5000 utilities that can be used to examine various files from the MS-DOS command line. These programs are provided with DSP/BIOS in the bin subdirectory. Any other utilities that may occasionally reside in the bin subdirectory and not documented here are for internal Texas Instruments' use only.

Topic	Page
3.1 cdbprint .....	3-2
3.2 gconfgen .....	3-3
3.3 nmti .....	3-7
3.4 sectti .....	3-8
3.5 sizeti .....	3-9
3.6 vers .....	3-10

### 3.1 cdbprint

*Prints a listing of all parameters defined in a configuration file*

#### Syntax

`cdbprint [-a] [-l] [-w] cdb-file`

#### Description

This utility reads a .cdb file created with the Configuration Tool and creates a list of all the objects and parameters. This tool can be used to compare two configuration files or to simply review the values of a single configuration file.

The -a flag causes cdbprint to list all objects and fields including those that are normally not visible (i.e., unconfigured objects and hidden fields). Without this flag, cdbprint ignores unconfigured objects or modules as well as any fields that are hidden.

The -l flag causes cdbprint to list the internal parameter names instead of the labels used by the Configuration Tool. Without this flag, cdbprint lists the labels used by the Configuration Tool.

The -w flag causes cdbprint to list only those parameters that can also be modified in the Configuration Tool. Without this flag, cdbprint lists both read-only and read-write parameters.

#### Example

The following sequence of commands can be used to compare a configuration file called test62.cdb to the default configuration provided with DSP/BIOS:

```
cdbprint ../../include/bios62.cdb > original.txt
cdbprint test62.cdb > test62.txt
diff original.txt test62.txt
```

## 3.2 gconfgen

*Reads a .cdb file created with the Configuration Tool*

### Syntax

gconfgen cdb-file

### Description

This command line utility reads a .cdb file (e.g. program.cdb) created with the Configuration Tool, where program is the name of your project, or program. The utility generates the target configuration files that are linked with the rest of the application code.

When you save a configuration file, the following files are created.

- ❑ **program.cdb.** Stores configuration settings for use by the Configuration Tool
- ❑ **programcfg.cmd.** Linker command file
- ❑ **programcfg.h54.** Assembly language header file included by hellocfg.s55
- ❑ **programcfg.h55.** Assembly language header file included by hellocfg.s55
- ❑ **programcfg.s54.** Assembly language source file
- ❑ **programcfg.s55.** Assembly language source file
- ❑ **programcfg\_c.c.** Source file to define Chip Support Library (CSL) structures and properties. See the CSL documentation for more information.
- ❑ **programcfg.h.** Header file to include CSL header files and declare external variables for CSL objects. See the CSL documentation for more information.

This utility is useful when the build process is controlled by a scripted mechanism, such as a make file, to generate the configuration source files from the configuration database file (.cdb file). Caution should be used, however, following product upgrades, since gconfgen does not detect revision changes. After a product update, use the graphical Configuration Tool to update your .cdb files to the new version. Once updated, gconfgen can be used again to generate the target configuration files.

## Example



You can use gconfgen, as shown in the following example, from the makefiles provided with the DSP/BIOS examples in the product distribution. To use gconfgen from the command line or makefiles, use its full path (TI\_DIR\plugins\bios\gconfgen) or add its folder (TI\_DIR\plugins\bios) to your PATH environment variable. (Note that TI\_DIR is the root directory of the product distribution).

```
*
* Makefile for creation of program named by the
* PROG variable
*
* These conventions are used in this makefile:
*   <prog>.asm - C54 assembly language source file
*   <prog>.obj - C54 object file (compiles/assembled)
*   <prog>.out - C54 executable (fully linked program)
*   <prog>cfg.s54 - configuration assembly source file
*                   generated by Configuration Tool
*   <prog>cfg.h54 - configuration assembly header file
*                   generated by Configuration Tool
*   <prog>cfg.cmd - configuration linker command file
*                   generated by Configuration Tool
*
TI_DIR := $(subst \,/,$(TI_DIR))
include $(TI_DIR)/c5400/bios/include/c54rules.mak
*
* Compiler, assembler, and linker options.
*
* -g enable symbolic debugging

CC54OPTS = -g
AS54OPTS =
* -q quiet run
LD54OPTS = -q      * -q quiet run
*
* Every BIOS program must be linked with:
*   $(PROG)cfg.o54 - from assembling $(PROG)cfg.s54
*   $(PROG)cfg.cmd - linker command file
*                   generated by Configuration Tool
*   If additional linker command files exist,
*   $(PROG)cfg.cmd must appear first.
*
PROG    = tsktest
OBJS    = $(PROG)cfg.obj
LIBS    =
CMDS    = $(PROG)cfg.cmd
```

```

*
*   Targets:
*
all:: $(PROG).out
$(PROG).out: $(OBJS) $(CMDS)
$(PROG)cfg.obj: $(PROG)cfg.h54
$(PROG).obj:
$(PROG)cfg.s54 $(PROG)cfg.h54 $(PROG)cfg.cmd ::
$(PROG).cdb      $(TI_DIR)/plugins/bios/gconfgcn
$(PROG).cdb
.clean clean::
    @ echo removing generated configuration files ...
    @$ (REMOVE) -f $(PROG)cfg.s54 $(PROG)cfg.h54
$(PROG)cfg.cmd
    @ echo removing object files and binaries ...
    @$ (REMOVE) -f *.obj *.out *.lst *.map

```



```

*
*   Makefile for creation of program named by the
*   PROG variable
*
*   These naming conventions are used by this makefile:
*   <prog>.asm - C55 assembly language source file
*   <prog>.obj - C55 object file (compiled/assembled)
*   <prog>.out - C55 executable (fully linked program)
*   <prog>cfg.s55 - configuration assembly source file
*                   generated by Configuration Tool
*   <prog>cfg.h55 - configuration assembly header file
*                   generated by Configuration Tool
*   <prog>cfg.cmd - configuration linker command file
*                   generated by Configuration Tool
*

```

```

TI_DIR := $(subst \,/,$(TI_DIR))
include $(TI_DIR)/c5500/bios/include/c55rules.mak
*
*   Compiler, assembler, and linker options.
*
* -g enable symbolic debugging

```

```

CC55OPTS = -g
AS55OPTS =
* -q quiet run
LD55OPTS = -q      * -q quiet run

```



```
*
* Every BIOS program must be linked with:
* $(PROG)cfg.o55 - from assembling $(PROG)cfg.s55
* $(PROG)cfg.cmd - linker command file
*                      generated by Configuration Tool
* If additional linker command files exist,
* $(PROG)cfg.cmd must appear first.
*
PROG    = tsctest
OBJS    = $(PROG)cfg.obj
LIBS    =
CMDS    = $(PROG)cfg.cmd
*
* Targets:
*
all:: $(PROG).out
$(PROG).out: $(OBJS) $(CMDS)
$(PROG)cfg.obj: $(PROG)cfg.h55
$(PROG).obj:
$(PROG)cfg.s55 $(PROG)cfg.h55 $(PROG)cfg.cmd ::
$(PROG).cdb    $(TI_DIR)/plugins/bios/gconfgen
$(PROG).cdb
.clean clean::
    @ echo removing generated configuration files ...
    @$(REMOVE) -f $(PROG)cfg.s55 $(PROG)cfg.h55
$(PROG)cfg.cmd
    @ echo removing object files and binaries ...
    @$(REMOVE) -f *.obj *.out *.lst *.map
```

### 3.3 nmti

*Display symbols and values in a TI COFF file*

#### Syntax

nmti [file1 file2 ...]

#### Description

nmti prints the symbol table (name list) for each TI executable file listed on the command line. Executable files must be stored as COFF (Common Object File Format) files.

If no files are listed, the file a.out is searched. The output is sent to stdout. Note that both linked (executable) and unlinked (object) files can be examined with nmti.

Each symbol name is preceded by its value (blanks if undefined) and one of the following letters:

A	absolute symbol
B	bss segment symbol
D	data segment symbol
E	external symbol
S	section name symbol
T	text segment symbol
U	undefined symbol

The letter is upper case if the symbol is external, and lower case if it is local.

### 3.4 sectti

*Display information about sections in TI COFF files*

#### Syntax

```
sectti [-a] [file1 file2 ...]
```

#### Description

sectti displays location and size information for all the sections in a TI executable file. Executable files must be stored as COFF (Common Object File Format) files.

All values are in hexadecimal. If no file names are given, a.out is assumed. Note that both linked (executable) and unlinked (object) files can be examined with sectti.

Using the -a flag causes sectti to display all program sections, including sections used only on the target by the DSP/BIOS plug-ins. If you omit the -a flag, sectti displays only the program sections that are loaded on the target.

### 3.5 sizeti

*Display the section sizes of an object file*

#### Syntax

sizeti[file1 file2 ...]

#### Description

This utility prints the decimal number of MADUs required by all code sections, all data sections, and the .bss and .stack sections for each COFF file argument. If no file is specified, a.out is used. Note that both linked (executable) and unlinked (object) files can be examined with this utility.

All sections that are located in program memory, including the .cinit data initialization section, are included as part of the value reported by the sizeti utility. Depending on how the executable is built, the .cinit section may not require space on the DSP.

Here is example output from the sizeti utility:

```
>cd \ti\tutorial\dsk5402\hello2\Debug
>c:\ti\bin\sizeti hello.out
   code    data    bss+stack    total
   6357     3311        591      10259    hello.out

>cd \ti\tutorial\evm5510\hello2\Debug
>c:\ti\bin\sizeti hello.out
   code    data    bss+stack    total
   82653     468        2622    85743    hello.out
```

---

## 3.6 vers

*Display version information for a DSP/BIOS source or library file*

### Syntax

```
vers [file1 file2 ...]
```

### Description

The vers utility displays the version number of DSP/BIOS files installed in your system. For example, the following command checks the version number of the bios.a54 or bios.a55 file in the lib sub-directory.

```
..\bin\vers bios.a54
bios.a54:
*** library
*** "date and time"
*** bios-c06
*** "version number"
```

or

```
..\bin\vers bios.a55
bios.a55:
*** library
*** "date and time"
*** bios-c06
*** "version number"
```

The actual output from vers may contain additional lines of information. To identify your software version number to Technical Support, use the version number shown.

Note that both libraries and source files can be examined with vers.

# Function Callability and Error Tables

---

---

---

This appendix provides tables describing TMS320C5000™ errors and function callability.

Topic	Page
A.1 Functions Callable by Tasks, SWI Handlers, or Hardware ISRs . . .	A-2
A.2 DSP/BIOS Error Codes . . . . .	A-8

## A.1 Functions Callable by Tasks, SWI Handlers, or Hardware ISRs

Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
ATM_andi	C	Yes	Yes	Yes	No
ATM_andu	C	Yes	Yes	Yes	No
ATM_cleari	C	Yes	Yes	Yes	No
ATM_clearu	C	Yes	Yes	Yes	No
ATM_deci	C	Yes	Yes	Yes	No
ATM_decu	C	Yes	Yes	Yes	No
ATM_inci	C	Yes	Yes	Yes	No
ATM_incu	C	Yes	Yes	Yes	No
ATM_ori	C	Yes	Yes	Yes	No
ATM_oru	C	Yes	Yes	Yes	No
ATM_seti	C	Yes	Yes	Yes	No
ATM_setu	C	Yes	Yes	Yes	No
C54_disableIMR	C	Yes	Yes	Yes	No
C54_enableIMR	C	Yes	Yes	Yes	No
C54_plug	C	Yes	Yes	Yes	No
C55_disableIMR0[IMR1]	C, assembly	Yes	Yes	Yes	No
C55_enableIMR0[IMR1]	C, assembly	Yes	Yes	Yes	No
C55_plug	C	Yes	Yes	Yes	No
CLK_countspms	C, assembly	Yes	Yes	Yes	No
CLK_gethtime	C, assembly	Yes	Yes	Yes	No
CLK_getltime	C, assembly	Yes	Yes	Yes	No
CLK_getprd	C, assembly	Yes	Yes	Yes	No
DEV_match	C	Yes	Yes	Yes	No
HST_getpipe	C, assembly	Yes	Yes	Yes	No
HWI_disable	C, assembly	Yes	Yes	Yes	No
HWI_dispatchPlug	none	Yes	Yes	Yes	No
HWI_enable	C, assembly	Yes	Yes	Yes	Yes*
HWI_enter	assembly	No	No	Yes	No
HWI_exit	assembly	No	No	Yes	Yes

Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
HWI_restore	C, assembly	Yes	Yes	Yes	Yes*
IDL_run	C	Yes	No	No	No
LCK_create	C	Yes	No	No	Yes*
LCK_delete	C	Yes	No	No	Yes*
LCK_pend	C	Yes	Yes*	Yes*	Yes*
LCK_post	C	Yes	Yes*	Yes*	Yes*
LOG_disable	C, assembly	Yes	Yes	Yes	No
LOG_enable	C, assembly	Yes	Yes	Yes	No
LOG_error	C, assembly	Yes	Yes	Yes	No
LOG_event	C, assembly	Yes	Yes	Yes	No
LOG_message	C, assembly	Yes	Yes	Yes	No
LOG_printf	C, assembly	Yes	Yes	Yes	No
LOG_reset	C, assembly	Yes	Yes	Yes	No
MBX_create	C	Yes	No	No	Yes*
MBX_delete	C	Yes	No	No	Yes*
MBX_pend	C	Yes	Yes*	Yes*	Yes*
MBX_post	C	Yes	Yes*	Yes*	Yes*
MEM_alloc	C	Yes	No	No	Yes*
MEM_calloc	C	Yes	No	No	Yes*
MEM_define	C	No	No	No	No*
MEM_free	C	Yes	No	No	Yes*
MEM_redefine	C	No	No	No	No*
MEM_stat	C	Yes	No	No	Yes*
MEM_valloc	C	Yes	No	No	Yes*
PIP_alloc	C, assembly	Yes	Yes	Yes	Yes
PIP_free	C, assembly	Yes	Yes	Yes	Yes
PIP_get	C, assembly	Yes	Yes	Yes	Yes
PIP_getReaderAddr	C	Yes	Yes	Yes	No
PIP_getReaderNumFrames	C	Yes	Yes	Yes	No
PIP_getReaderSize	C	Yes	Yes	Yes	No
PIP_getWriterAddr	C	Yes	Yes	Yes	No
PIP_getWriterNumFrames	C	Yes	Yes	Yes	No



Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
PIP_getWriterSize	C	Yes	Yes	Yes	No
PIP_peek	C	Yes	Yes	Yes	No
PIP_put	C, assembly	Yes	Yes	Yes	Yes
PIP_reset	C	Yes	Yes	Yes	Yes
PIP_setWriterSize	C	Yes	Yes	Yes	No
PRD_getticks	C, assembly	Yes	Yes	Yes	No
PRD_start	C, assembly	Yes	Yes	Yes	No
PRD_stop	C, assembly	Yes	Yes	Yes	No
PRD_tick	C, assembly	Yes	Yes	Yes	Yes
QUE_create	C	Yes	No	No	Yes*
QUE_delete	C	Yes	No	No	Yes*
QUE_dequeue	C	Yes	Yes	Yes	No
QUE_empty	C	Yes	Yes	Yes	No
QUE_enqueue	C	Yes	Yes	Yes	No
QUE_get	C	Yes	Yes	Yes	No
QUE_head	C	Yes	Yes	Yes	No
QUE_insert	C	Yes	Yes	Yes	No
QUE_new	C	Yes	Yes	Yes	No
QUE_next	C	Yes	Yes	Yes	No
QUE_prev	C	Yes	Yes	Yes	No
QUE_put	C	Yes	Yes	Yes	No
QUE_remove	C	Yes	Yes	Yes	No
RTDX_channelBusy	C, assembly	Yes	Yes	No	No
RTDX_CreateInputChannel	C, assembly	Yes	Yes	No	No
RTDX_CreateOutputChannel	C, assembly	Yes	Yes	No	No
RTDX_disableInput	C, assembly	Yes	Yes	No	No
RTDX_disableOutput	C, assembly	Yes	Yes	No	No
RTDX_enableInput	C, assembly	Yes	Yes	No	No
RTDX_enableOutput	C, assembly	Yes	Yes	No	No
RTDX_isInputEnabled	C, assembly	Yes	Yes	No	No
RTDX_isOutputEnabled	C, assembly	Yes	Yes	No	No
RTDX_read	C, assembly	Yes	Yes	No	No

Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
RTDX_readNB	C, assembly	Yes	Yes	No	No
RTDX_sizeofInput	C, assembly	Yes	Yes	No	No
RTDX_write	C, assembly	Yes	Yes	No	No
SEM_count	C	Yes	Yes	Yes	No
SEM_create	C	Yes	No	No	Yes*
SEM_delete	C	Yes	No	No	Yes*
SEM_ipost	C	No	Yes	Yes	No
SEM_new	C	Yes	Yes	Yes	No
SEM_pend	C	Yes	Yes*	Yes*	Yes*
SEM_post	C	Yes	Yes	Yes	Yes*
SEM_reset	C	Yes	No	No	No
SIO_bufsize	C	Yes	Yes	Yes	No
SIO_create	C	Yes	No	No	Yes*
SIO_ctrl	C	Yes	No	No	No
SIO_delete	C	Yes	No	No	Yes*
SIO_flush	C	Yes	No	No	No
SIO_get	C	Yes	No	No	Yes*
SIO_idle	C	Yes	No	No	Yes*
SIO_issue	C	Yes	No	No	No
SIO_put	C	Yes	No	No	Yes*
SIO_reclaim	C	Yes	No	No	Yes*
SIO_segid	C	Yes	Yes	Yes	No
SIO_select	C	Yes	No	No	Yes*
SIO_staticbuf	C	Yes	No	No	No
STS_add	C, assembly	Yes	Yes	Yes	No
STS_delta	C, assembly	Yes	Yes	Yes	No
STS_reset	C, assembly	Yes	Yes	Yes	No
STS_set	C, assembly	Yes	Yes	Yes	No
SWI_andn	C, assembly	Yes	Yes	Yes	Yes*
SWI_andnHook	C, assembly	Yes	Yes	Yes	Yes*
SWI_create	C	Yes	No	No	Yes*
SWI_dec	C, assembly	Yes	Yes	Yes	Yes*

Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
SWI_delete	C	Yes	No	No	Yes*
SWI_disable	C, assembly	Yes	Yes	No	No
SWI_enable	C, assembly	Yes	Yes	No	Yes*
SWI_getattr	C	Yes	Yes	Yes	No
SWI_getmbox	C, assembly	No	Yes	No	No
SWI_getpri	C, assembly	Yes	Yes	Yes	No
SWI_inc	C, assembly	Yes	Yes	Yes	Yes*
SWI_or	C, assembly	Yes	Yes	Yes	Yes*
SWI_orHook	C, assembly	Yes	Yes	Yes	Yes*
SWI_post	C, assembly	Yes	Yes	Yes	Yes*
SWI_raisepri	C, assembly	No	Yes	No	No
SWI_restorepri	C, assembly	No	Yes	No	Yes
SWI_self	C, assembly	No	Yes	No	No
SWI_setattr	C	Yes	Yes	Yes	No
SYS_abort	C	Yes	Yes	Yes	No
SYS_atexit	C	Yes	Yes	Yes	No
SYS_error	C	Yes	Yes	Yes	No
SYS_exit	C	Yes	Yes	Yes	No
SYS_printf	C	Yes	Yes	Yes	No
SYS_putchar	C	Yes	Yes	Yes	No
SYS_sprintf	C	Yes	Yes	Yes	No
SYS_vprintf	C	Yes	Yes	Yes	No
SYS_vsprintf	C	Yes	Yes	Yes	No
TRC_disable	C, assembly	Yes	Yes	Yes	No
TRC_enable	C, assembly	Yes	Yes	Yes	No
TRC_query	C, assembly	Yes	Yes	Yes	No
TSK_checkstacks	C	Yes	No	No	No
TSK_create	C	Yes	No	No	Yes*
TSK_delete	C	Yes	No	No	Yes*
TSK_deltatime	C	Yes	Yes	Yes	No
TSK_disable	C	Yes	No	No	No
TSK_enable	C	Yes	No	No	Yes*

Function	Interface (C and/or Assembly)	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
TSK_exit	C	Yes	No	No	Yes*
TSK_getenv	C	Yes	Yes	Yes	No
TSK_geterr	C	Yes	Yes	Yes	No
TSK_getname	C	Yes	Yes	Yes	No
TSK_getpri	C	Yes	Yes	Yes	No
TSK_getsts	C	Yes	Yes	Yes	No
TSK_itick	C	No	Yes	Yes	Yes
TSK_self	C	Yes	Yes	Yes	No
TSK_setenv	C	Yes	Yes	Yes	No
TSK_seterr	C	Yes	Yes	Yes	No
TSK_setpri	C	Yes	Yes	Yes	Yes*
TSK_settime	C	Yes	Yes	Yes	No
TSK_sleep	C	Yes	No	No	Yes*
TSK_stat	C	Yes	Yes*	Yes*	No
TSK_tick	C	Yes	Yes	Yes	Yes*
TSK_time	C	Yes	Yes	Yes	No
TSK_yield	C	Yes	Yes	Yes	Yes*

Note: \*See the appropriate API reference page for more information.

## A.2 DSP/BIOS Error Codes

Name	Value	SYS_Errors[Value]
SYS_OK	0	"(SYS_OK)"
SYS_EALLOC	1	"(SYS_EALLOC): segid = %d, size = %u, align = %u" Memory allocation error.
SYS_EFREE	2	"(SYS_EFREE): segid = %d, ptr = 0x%x, size = %u" The memory free function associated with the indicated memory segment was unable to free the indicated size of memory at the address indicated by ptr.
SYS_ENODEV	3	"(SYS_ENODEV): device not found" The device being opened is not configured into the system.
SYS_EBUSY	4	"(SYS_EBUSY): device in use" The device is already opened by the maximum number of users.
SYS_EINVAL	5	"(SYS_EINVAL): invalid parameter" An invalid parameter was passed to the device.
SYS_EBADIO	6	"(SYS_EBADIO): device failure" The device was unable to support the I/O operation.
SYS_EMODE	7	"(SYS_EMODE): invalid mode" An attempt was made to open a device in an improper mode; e.g., an attempt to open an input device for output.
SYS_EDOMAIN	8	"(SYS_EDOMAIN): domain error" Used by SPOX-MATH when type of operation does not match vector or filter type.
SYS_ETIMEOUT	9	"(SYS_ETIMEOUT): timeout error" Used by device drivers to indicate that reclaim timed out.
SYS_EEOF	10	"(SYS_EEOF): end-of-file error" Used by device drivers to indicate the end of a file.
SYS_EDEAD	11	"(SYS_EDEAD): previously deleted object" An attempt was made to use an object that has been deleted.
SYS_EBADOBJ	12	"(SYS_EBADOBJ): invalid object" An attempt was made to use an object that does not exist.
SYS_EUSER	>=256	"(SYS_EUSER): <user-defined string>" User-defined error.

# C55x DSP/BIOS Register Use and Preservation Conventions

---

---

---

---

This appendix provides tables describing the TMS320C55x™ register conventions in terms of usage and preservation across multi-threaded context switching.

Topic	Page
B.1 Preservation Model for Non-Status CPU Registers .....	B-2
B.2 Preservation Model for Procesor Status Registers .....	B-5

## B.1 Preservation Model for Non-Status CPU Registers

Table B.1 specifies how the non-status registers are treated within the DSP/BIOS context.

Within the heading Register Usage, the columns headed C Compiler and DSP/BIOS, you can find the following terms:

- ☐ **Caller preserved.** The compiler requires that the calling function preserve the register.
- ☐ **Callee preserved.** The compiler requires that the called function preserve the register.
- ☐ **Unmodified.** Neither the C compiler nor the DSP/BIOS APIs use or modify this register.
- ☐ **Modified.** The C compiler or the DSP/BIOS APIs use and can modify this register.

The heading Register Preservation Model indicates the register usage in the HWI, SWI and TSK schedulers. The usage terms are:

- ☐ **Preserved.** The named schedulers maintain the register as context of the thread.
- ☐ **Not Preserved.** The named schedulers do not maintain the register as context of the thread.

Register Name	Register Usage		Register Preservation Model	
	C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
AC0	Caller preserved	Modified	HWI, SWI	TSK
AC1	Caller preserved	Modified	SWI	HWI, TSK
AC2	Caller preserved	Modified	SWI	HWI, TSK
AC3	Caller preserved	Modified	SWI	HWI, TSK
BK03	Unmodified	Unmodified	SWI	HWI, TSK
BK47	Unmodified	Unmodified	SWI	HWI, TSK
BKC	Unmodified	Unmodified	SWI	HWI, TSK
BRC0	Unmodified	Unmodified	SWI	HWI, TSK
BRC1	Unmodified	Unmodified	SWI	HWI, TSK
BRS1	Unmodified	Unmodified	SWI	HWI, TSK

Register Name	Register Usage		Register Preservation Model	
	C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
BSA01	Unmodified	Unmodified	SWI	HWI, TSK
BSA23	Unmodified	Unmodified	SWI	HWI, TSK
BSA45	Unmodified	Unmodified	SWI	HWI, TSK
BSA67	Unmodified	Unmodified	SWI	HWI, TSK
BSAC	Unmodified	Unmodified	SWI	HWI, TSK
CDP	Unmodified	Unmodified	SWI	HWI, TSK
CFCT	Callee preserved	Unmodified	SWI	HWI, TSK
CSR	Caller preserved	Modified	SWI	HWI, TSK
DP	Unmodified	Unmodified	SWI	HWI, TSK
DBIER0	Unmodified	Unmodified	—	HWI, SWI, TSK
DBIER1	Unmodified	Unmodified	—	HWI, SWI, TSK
ISTR	Unmodified	Unmodified	—	HWI, SWI, TSK
ICR	Unmodified	Unmodified	—	HWI, SWI, TSK
IER0	Unmodified	Modified	HWI	SWI, TSK
IER1	Unmodified	Modified	HWI	SWI, TSK
IFR0,	Unmodified	Modified	—	HWI, SWI, TSK
IFR1	Unmodified	Modified	—	HWI, SWI, TSK
IVPD	Unmodified	Modified	—	HWI, SWI, TSK
IVPH	Unmodified	Modified	—	HWI, SWI, TSK
PDP	Unmodified	Unmodified	—	HWI, SWI, TSK
REA0	Caller preserved	Modified	SWI	HWI, TSK
REA1	Caller preserved	Unmodified	SWI	HWI, TSK
RETA	Callee preserved	Unmodified	TSK	HWI, SWI
RPTC	Caller preserved	Modified	SWI	HWI, TSK
RSA0	Caller preserved	Modified	SWI	HWI, TSK



Register Name	Register Usage		Register Preservation Model	
	C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
RSA1	Caller preserved	Unmodified	SWI	HWI, TSK
T0	Caller preserved	Modified	SWI	HWI, TSK
T1	Caller preserved	Modified	SWI	HWI, TSK
T2	Callee preserved	Modified	TSK	HWI, SWI
T3	Callee preserved	Modified	TSK	HWI, SWI
TRN0	Unmodified	Unmodified	SWI	HWI, TSK
TRN1	Unmodified	Unmodified	SWI	HWI, TSK
XAR0	Caller preserved	Modified	HWI, SWI	TSK
XAR1	Caller preserved	Modified	HWI, SWI	TSK
XAR2	Caller preserved	Modified	SWI	HWI, TSK
XAR3	Caller preserved	Modified	SWI	HWI, TSK
XAR4	Caller preserved	Modified	SWI, TSK	HWI
XAR5	Callee preserved	Unmodified	SWI, TSK	HWI
XAR6	Callee preserved	Unmodified	SWI, TSK	HWI
XAR7	Callee preserved	Unmodified	SWI, TSK	HWI

## B.2 Preservation Model for Processor Status Registers

Table B.2 specifies how the status registers ST0, ST1, ST2, and ST3 are treated within the DSP/BIOS context.

When you do not alter the boot time settings, the operating system ensures that the environment is preserved. You do not need to do any additional processing. This is normally the case when DSP/BIOS APIs are called asynchronously as in interrupts.

For synchronous entry into a DSP/BIOS API, the settings listed in the DSP/BIOS column of Table B.2 become a precondition for calling the API. Typically, this is not an issue for C or C++ applications.

DSP/BIOS provides utility macros to assist you in providing the necessary preconditions in Assembly language applications. The definitions of these macros can be found in the `c55.h55` include file. The macros are:

- ❑ **C55\_setBiosSTbits.** This macro sets processor Status register bits as required by DSP/BIOS.
- ❑ **C55\_saveBiosContext.** This macro saves on the stack all registers declared as Preserved in the DSP/BIOS multi-threading context.
- ❑ **C55\_restoreBiosContext.** This macro restores from the stack all registers declared as Preserved in the DSP/BIOS multi-threading context.
- ❑ **C55\_saveCcontext.** This macro saves on the stack all registers declared as Caller preserved in the C compiler conventions.
- ❑ **C55\_restoreCcontext.** This macro restores from the stack all registers declared as Caller preserved in the C compiler conventions.

Table B.2 includes the following columns:

- ❑ **Status Bit.** This column gives the name of the status bit in the form <register name>\_<bit name>.
- ❑ **Type.** This attribute specifies the type of the status bit. The values are:
  - **Local.** The bit is thread-specific, meaning that each thread has its own copy of this status bit and it is a part of the thread's context.
  - **Global.** The bit controls aspects of the processor and the bit is not part of the thread's context.
- ❑ **Status Bit Mandatory Settings.** The two columns under this heading list the value assumed by C compiler and the DSP/BIOS APIs. The values in the column DSP/BIOS APIs are set during system initialization or bootup by DSP/BIOS. The values are:
  - **None.** No assumption is made on this bit.
  - **1.** A value of 1 is a prerequisite in this bit position
  - **0.** A value of 0 is a prerequisite in this bit position.
- ❑ **Register Preservation Model.** These two columns indicate the register usage in the HWI, SWI and TSK schedulers. The usage terms are:
  - **Preserved.** The named schedulers maintain the Status Bit setting as context of the thread.
  - **Not Preserved.** The named schedulers do not maintain the Status Bit setting as context of the thread.

Status Bit	Type	Status Bit Mandatory Settings		Register Preservation Model	
		C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
ST0_AC0V2	Local	None	None	HWI, SWI, TSK	–
ST0_AC0V3	Local	None	None	HWI, SWI, TSK	–
ST0_TC1	Local	None	None	HWI, SWI, TSK	–
ST0_TC2	Local	None	None	HWI, SWI, TSK	–
ST0_CARRY	Local	None	None	HWI, SWI, TSK	–

Status Bit	Type	Status Bit Mandatory Settings		Register Preservation Model	
		C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
ST0_AC0V0	Local	None	None	HWI, SWI, TSK	–
ST0_ACOV1	Local	None	None	HWI, SWI, TSK	–
ST0_DP	Local	None	None	HWI, SWI, TSK	–
ST1_BRAF	Local	None	0	HWI, SWI, TSK	–
ST1_CPL	Local	1	1	HWI, SWI, TSK	–
ST1_XF	Global	None	None	HWI	SWI, TSK
ST1_HM	Global	None	None	HWI	SWI, TSK
ST1_INTM	Global	None	None	HWI, SWI, TSK	–
ST1_M40	Local	0	0	HWI, SWI, TSK	–
ST1_SATD	Local	0	0	HWI, SWI, TSK	–
ST1_SXMD	Local	1	1	HWI, SWI, TSK	–
ST1_C16	Local	None	0	HWI, SWI, TSK	–
ST1_FRCT	Local	0	0	HWI, SWI, TSK	–
ST1_C54CM	Local	0	0	HWI, SWI, TSK	–
ST2_ARMS	Local	1	1	HWI, SWI, TSK	–
ST2_DBGM	Global	None	None	HWI	SWI, TSK
ST2_EALLOW	Global	None	None	HWI	SWI, TSK
ST2_RDM	Local	0	0	HWI, SWI, TSK	–
ST2_CDPLC	Local	0	0	HWI, SWI, TSK	–
ST2_AR0-7LC	Local	0	0	HWI, SWI, TSK	–
ST3_CARFRZ	Global	None	None	–	HWI, SWI, TSK
ST3_CAEN	Global	None	None	–	HWI, SWI, TSK
ST3_CACLR	Global	None	None	–	HWI, SWI, TSK
ST3_HINT	Global	None	None	–	HWI, SWI, TSK
ST3_CBERR	Global	None	None	–	HWI, SWI, TSK

Status Bit	Type	Status Bit Mandatory Settings		Register Preservation Model	
		C Compiler	DSP/BIOS APIs	Preserved	Not Preserved
ST3_MPNMC	Global	None	None	–	HWI, SWI, TSK
ST3_SATA	Local	0	0	HWI, SWI, TSK	–
ST3_CLKOFF	Global	None	None	–	HWI, SWI, TSK
ST3_SMUL	Local	0	0	HWI, SWI, TSK	–
ST3_SST	Local	None	0	HWI, SWI, TSK	–

32-bit register 1-3

## A

arg 2-52, 2-57  
assembly  
    time 2-2  
assembly language  
    calling C functions from 2-3  
atexit 2-356  
ATM\_andi 2-5  
ATM\_andu 2-6  
ATM\_cleari 2-7  
ATM\_clearu 2-8  
ATM\_deci 2-9  
ATM\_decu 2-10  
ATM\_inci 2-11  
ATM\_incu 2-12  
ATM\_ori 2-13  
ATM\_oru 2-14  
ATM\_seti 2-15  
ATM\_setu 2-16  
atomic operations 2-181  
atomic queue 2-181  
average 2-244

## B

background loop 2-103  
bind 2-51  
boards  
    setting 2-74  
BSCR 2-75  
buffered pipe manager 2-148

## C

C functions  
    calling from assembly language 2-3  
C54 Module 2-17  
C54\_disable  
    main description 2-18

C54\_disableIMR 2-17, 2-244  
C54\_enableIMR 2-24  
C54\_plug  
    main description 2-24  
C55 Module 2-17  
C55\_disable  
    main description 2-19  
C55\_plug  
    main description 2-25  
Call User Init Function property 2-75  
calloc 2-356  
cdb files 3-3  
cdbprint utility 3-2  
channels 2-76  
Chip Support Library 2-74  
Chip Support Library name 2-74  
CLK - Code Composer Studio Interface 2-28  
CLK module 2-26  
    global properties 2-27  
    trace types 2-312  
CLK Object Properties 2-28  
CLK\_countspms 2-29  
CLK\_gethetime 2-31, 2-34  
CLK\_gettime 2-33  
CLK\_getprd 2-26, 2-35  
CLKMD - (PLL) Clock Mode Register 2-75  
Clock Manager Properties 2-27  
clocks  
    real time vs. data-driven 2-169  
comment 2-28, 2-38, 2-62, 2-68  
configuration files 3-3  
    printing 3-2  
Configuration Tool 2-3  
context switch 2-3  
conversion specifications 2-303, 2-305, 2-307, 2-309  
count 2-244  
counts per millisecond 2-29  
create function 2-57  
ctrl 2-51

## D

data channels 2-76

- data transfer 2-148
- DAX 2-38
- DAX driver 2-50
- dax.h 2-50
- delete function 2-57
- den 2-57
- DEV Manager Properties 2-37
- DEV module 2-36
- DEV Object Properties 2-38
- DEV\_Fxns 2-37
- DEV\_FXNS table 2-50, 2-56, 2-63, 2-64, 2-69, 2-71
- DEV\_Fxns table 2-38
- DEV\_match 2-39
- device driver interface 2-36
- Device ID 2-38, 2-50, 2-56, 2-63, 2-64, 2-69, 2-71
- device object
  - user-defined 2-38
- device table 2-39
- devices
  - empty 2-63
- DGN 2-37
- DGN driver 2-53
- DGS 2-38
- DGS driver 2-56
- dgs.h 2-56
- DHL 2-37
- DHL driver 2-60
- DHL Driver Properties 2-61
- DHL Object Properties 2-62
- Directly configure on-device timer registers 2-27
- disable
  - HWI 2-88, 2-90, 2-101
  - LOG 2-114
  - SWI 2-272
  - TRC 2-315
- disabling
  - hardware interrupts 2-88, 2-90, 2-101
- DNL 2-38
- DNL driver 2-63
- DOV 2-38
- DOV driver 2-64
- DPI 2-37
- DPI driver 2-66
- DPI Driver Properties 2-68
- DPI Object Properties 2-68
- drivers
  - DAX 2-50
  - DGN 2-53
  - DGS 2-56
  - DHL 2-60
  - DNL 2-63
  - DOV 2-64
  - DPI 2-66
  - DST 2-69
  - DTR 2-71
- DSP Speed In MHz (CLKOUT) 2-74

- DSP Type 2-74
- DST 2-38
- DST driver 2-69
- DTR 2-38
- DTR driver 2-71
- dtr.h 2-72
- Dxx 2-37
- Dxx\_close 2-40
- Dxx\_ctrl 2-41
  - error handling 2-41
- Dxx\_idle 2-42
  - error handling 2-42
- Dxx\_init 2-43
- Dxx\_issue 2-44
- Dxx\_open 2-46
- Dxx\_ready 2-47
- Dxx\_reclaim 2-44, 2-48
  - error handling 2-48

## E

- enable
  - HWI 2-91
  - LOG 2-115
  - SWI 2-274
  - TRC 2-317
- Enable All TRC Trace Event Classes 2-75
- Enable CLK Manager 2-27
- Enable Real Time Analysis 2-75
- enabling
  - hardware interrupts 2-91
- environ 2-356
- Error Codes A-8
- error handling
  - by Dxx\_close 2-40
  - by Dxx\_ctrl 2-41
  - by Dxx\_idle 2-42
  - by Dxx\_reclaim 2-48
- Execution Graph 2-28
- exit 2-356

## F

- files
  - .h 2-17
- Fix TDDR 2-28
- flush 2-42
- free 2-356
- function 2-28
- Function Call Model 2-75
- functions
  - list of 1-4

## G

gconfgn utility 3-3  
 generators 2-53  
 getenv 2-356  
 global settings 2-74  
 Global Settings Properties 2-74

## H

hardware interrupts 2-81  
   disabling 2-88, 2-90, 2-101  
   enabling 2-91  
 hardware timer counter register ticks 2-27  
 high-resolution time 2-26, 2-31, 2-32  
 host data interface 2-76  
 HST module 2-76  
 HST object  
   adding a new 2-60  
 HST\_getpipe 2-79  
 HWI module 2-81  
   statistics units 2-245  
   trace types 2-312  
 HWI\_disable 2-88, 2-90, 2-101  
   vs. instruction 2-2  
 HWI\_enable 2-91  
 HWI\_enter 2-93  
 HWI\_exit 2-97

## I

i16tof32/f32toi16 2-58  
 i16toi32/i32toi16 2-58  
 IDL module 2-103  
 IDL\_run 2-105  
 IER 2-19, 2-22  
 IER0 2-19  
 IMR 2-18, 2-20  
 Init Fxn 2-38, 2-50, 2-56, 2-63, 2-64, 2-69, 2-71  
 Instructions/Int 2-28  
 Interrupt Enable Register 2-19, 2-22  
 Interrupt Mask Register 2-18, 2-20  
 interrupt service routines 2-81  
 ISRs 2-81

## L

large memory model 2-3  
 LCK module 2-106  
 LCK\_create 2-107  
 LCK\_delete 2-108

LCK\_release 2-110  
 LCK\_seize 2-109  
 localcopy 2-58  
 LOG module 2-111  
 LOG\_disable 2-114  
 LOG\_enable 2-115  
 LOG\_error 2-116, 2-118  
 LOG\_event 2-120  
 LOG\_printf 2-122  
 LOG\_reset 2-126  
 logged events 2-312  
 low-resolution clock 2-26  
 low-resolution time 2-26, 2-32, 2-33

## M

mailbox  
   clear bits 2-262, 2-265  
   decrement 2-267, 2-269, 2-271, 2-276, 2-295  
   get value 2-278  
   increment 2-281  
   set bits 2-283, 2-285  
 malloc 2-356  
 mask 2-19  
 maximum 2-244  
 MBX module 2-127  
 MBX\_create 2-129  
 MBX\_delete 2-130  
 MBX\_pend 2-131  
 MBX\_post 2-132  
 MEM module 2-133  
 MEM\_alloc 2-141, 2-142, 2-147  
 MEM\_calloc 2-141, 2-142, 2-147  
 MEM\_define 2-143  
 MEM\_free 2-144  
 MEM\_NULL 2-136, 2-325  
 MEM\_redefine 2-145  
 MEM\_stat 2-146  
 MEM\_valloc 2-141, 2-142, 2-147  
 Memory Model 2-75  
 MHz 2-74  
 Microseconds/Int 2-27  
 Mode 2-62  
 Modifies  
   definition of 1-3  
 modifies 2-2  
 Modify CLKMD 2-75  
 modules  
   list of 1-2  
 multiprocessor application 2-68



## N

- naming conventions 1-3
- nmti utility 3-7
- notifyReader function
  - use of HWI\_enter 2-83
- ns 2-19
- num 2-57

## O

- Object Memory 2-27
- Object memory 2-61
- on-chip timer 2-26
- operations
  - list of 1-4
- outputdelay 2-52

## P

- packing/unpacking 2-56
- Parameters 2-38, 2-50, 2-56, 2-63, 2-64, 2-69, 2-71
- parameters
  - listing 3-2
  - vs. registers 2-2
- period register 2-35
- period register property 2-35
- PIP module 2-148
  - statistics units 2-245
- PIP\_alloc 2-152
- PIP\_free 2-154
- PIP\_get 2-156
- PIP\_getReaderAddr 2-158
- PIP\_getReaderNumFrames 2-159
- PIP\_getReaderSize 2-160
- PIP\_getWriterAddr 2-161
- PIP\_getWriterNumFrames 2-162
- PIP\_getWriterSize 2-163
- PIP\_put 2-164, 2-165, 2-167
- PIP\_setWriterSize 2-168
- pipe object 2-79
- pipes 2-148
- PMST(15-0) 2-74
- PMST(6-0) 2-74
- Postconditions
  - definition of 1-3
- postconditions 2-2
- posting software interrupts 2-259, 2-287
- PRD module 2-169
  - statistics units 2-245
  - trace types 2-312
- PRD register 2-28
- PRD\_getticks 2-172

- PRD\_start 2-174
- PRD\_stop 2-176
- PRD\_tick 2-178
- Preconditions 2-2
  - definition of 1-3
- preconditions 2-2
- printing configuration file 3-2
- priorities 2-260

## Q

- QUE module 2-180
- QUE\_create 2-182
- QUE\_delete 2-184
- QUE\_dequeue 2-185
- QUE\_empty 2-186
- QUE\_enqueue 2-187
- QUE\_get 2-188
- QUE\_head 2-189
- QUE\_insert 2-190
- QUE\_new 2-191
- QUE\_next 2-192
- QUE\_prev 2-193
- QUE\_put 2-194
- QUE\_remove 2-195

## R

- read data 2-149
- realloc 2-356
- registers
  - in Assembly Interface 1-3, 1-4
  - modified 2-2
  - vs. parameters 2-2
- RTA Control Panel 2-28
- RTDX Mode 2-197
- RTDX\_bytesRead 2-212
- RTDX\_channelBusy 2-199
- RTDX\_CreateInputChannel 2-200, 2-201
- RTDX\_CreateOutputChannel 2-200, 2-201
- RTDX\_disableInput 2-202, 2-203, 2-204, 2-205
- RTDX\_disableOutput 2-202, 2-203, 2-204, 2-205
- RTDX\_enableInput 2-202, 2-203, 2-204, 2-205
- RTDX\_enableOutput 2-202, 2-203, 2-204, 2-205
- RTDX\_isInputEnabled 2-206, 2-207
- RTDX\_isOutputEnabled 2-206, 2-207
- RTDX\_read 2-208
- RTDX\_readNB 2-210
- RTDX\_write 2-213

# S

sections  
   in executable file 3-8  
 sectti utility 3-8  
 SEM module 2-214  
 SEM\_count 2-216  
 SEM\_create 2-217  
 SEM\_delete 2-218  
 SEM\_ipost 2-219  
 SEM\_new 2-220  
 SEM\_pend 2-221  
 SEM\_post 2-222  
 SEM\_reset 2-223  
 signed integer  
   maximum 2-11  
   minimum 2-11  
 single-processor application 2-68  
 SIO module 2-224  
 SIO\_bufsize 2-227  
 SIO\_create 2-228  
 SIO\_ctrl 2-231  
 SIO\_delete 2-232  
 SIO\_flush 2-233  
 SIO\_get 2-234  
 SIO\_idle 2-235  
 SIO\_issue 2-236  
 SIO\_ISSUERECLAIM streaming model  
   and DPI 2-67  
 SIO\_put 2-238  
 SIO\_reclaim 2-48, 2-239  
 SIO\_segid 2-241  
 SIO\_select 2-47, 2-242, 2-243  
 sizeti utility 3-9  
 software interrupts 2-258  
 stack overflow check 2-328  
 stack, execution 2-260  
 Stacking Device 2-38, 2-49, 2-50, 2-56, 2-63, 2-64,  
   2-69, 2-71  
 start 2-51  
 statistics  
   units 2-245, 2-312  
 std.h 2-356  
 stdlib.h 2-356  
 stop 2-51  
 STS manager 2-197, 2-244  
 STS\_add 2-250  
 STS\_delta 2-252  
 STS\_reset 2-254  
 STS\_set 2-256  
 SWI module 2-258  
   statistics units 2-245  
   trace types 2-312  
 SWI\_andn 2-262, 2-265  
 SWI\_dec 2-267, 2-269, 2-271, 2-276, 2-295

SWI\_disable 2-272  
 SWI\_enable 2-274  
 SWI\_getmbx 2-278  
 SWI\_getpri 2-280  
 SWI\_inc 2-281  
 SWI\_or 2-283, 2-285  
 SWI\_post 2-287  
 SWI\_raisepri 2-289  
 SWI\_restorepri 2-291  
 SWI\_self 2-293  
 SWWSR 2-74  
 symbol table 3-7  
 SYS module 2-297  
 SYS\_abort 2-299  
 SYS\_atexit 2-300  
 SYS\_EALLOC A-8  
 SYS\_EBADIO A-8  
 SYS\_EBADOBJ A-8  
 SYS\_EBUSY A-8  
 SYS\_EDEAD A-8  
 SYS\_EDOMAIN A-8  
 SYS\_EEOF A-8  
 SYS\_EFREE A-8  
 SYS\_EINVAL A-8  
 SYS\_EMODE A-8  
 SYS\_ENODEV A-8  
 SYS\_error 2-141, 2-142, 2-147, 2-182, 2-217, 2-  
   228, 2-271, 2-301  
 SYS\_ETIMEOUT 2-48, A-8  
 SYS\_EUSER 2-301, A-8  
 SYS\_exit 2-302  
 SYS\_OK 2-40, A-8  
 SYS\_printf 2-303, 2-305, 2-307, 2-309  
 SYS\_putchar 2-311  
 system clock manager 2-26

# T

target board 2-74  
 task switch 2-20, 2-21, 2-23  
 tasks  
   on demand 2-53  
 TDDR 2-26  
 TDDR register 2-28  
 templates 2-37  
 timer 2-26  
 timer divid-down register 2-26  
 timer interrupt 2-34  
 total 2-244  
 trace types 2-312  
 transform function 2-56, 2-57  
 transformers 2-71  
 TRC module 2-312  
 TRC\_disable 2-315

- TRC\_enable 2-317
- TRC\_query 2-319
- TSK module 2-321
  - statistics units 2-245
- TSK\_checkstacks 2-328
- TSK\_create 2-329
- TSK\_delete 2-332
- TSK\_deltatime 2-334
- TSK\_disable 2-336
- TSK\_enable 2-337
- TSK\_exit 2-338
- TSK\_getenv 2-339
- TSK\_geterr 2-340
- TSK\_getname 2-341
- TSK\_getpri 2-342
- TSK\_getsts 2-343
- TSK\_itick 2-344
- TSK\_self 2-345
- TSK\_setenv 2-346
- TSK\_seterr 2-347
- TSK\_setpri 2-348
- TSK\_settime 2-349
- TSK\_sleep 2-351
- TSK\_stat 2-352
- TSK\_tick 2-353
- TSK\_time 2-354
- TSK\_yield 2-355

## U

- u16tou32/u32tou16 2-57
- u32tou8/u8tou32 2-57
- u8toi16/i16tou8 2-58

- unbind 2-51
- Underlying HST Channel 2-62
- underscore 2-28, 2-38
  - in function names 2-3
- units for statistics 2-245
- unsigned integer 2-12
  - maximum 2-10, 2-12
  - minimum 2-10
- Use high resolution time for internal timings 2-27
- User Init Function property 2-75
- USER traces 2-312
- utilities
  - cdbprint 3-2
  - gconfgen 3-3
  - nmti 3-7
  - sectti 3-8
  - size 3-9
  - vers 3-10

## V

- vecid 2-24, 2-25
- vector address 2-25
- vector table 2-24
- vers utility 3-10
- version information 3-10

## W

- write data 2-149