

# ***TMS320C6000 DSP/BIOS 5.x Application Programming Interface (API) Reference Guide***

Literature Number: SPRU403R  
November 2010



## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

### Products

Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
RF/IF and ZigBee® Solutions	<a href="http://www.ti.com/lprf">www.ti.com/lprf</a>

### Applications

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

# Read This First

---

---

---

### **About This Manual**

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320C6000™ 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 *Code Composer Studio* online 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, 62 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 .h62. For the C64x or C67x DSP platform, substitute either 64 or 67 for each occurrence of 62.
- ❑ 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.

***TMS320 DSP/BIOS User's Guide*** (literature number SPRU423) provides an overview and description of the DSP/BIOS real-time operating system.

***TMS320C6000 Optimizing C Compiler User's Guide*** (literature number SPRU187) describes the c6000 C/C++ compiler and the assembly optimizer. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the C6000 generation of devices.

***TMS320C6000 Programmer's Guide*** (literature number SPRU189) describes the c6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

***TMS320c6000 Peripherals Reference Guide*** (literature number SPRU190) describes common peripherals available on the TMS320C6000 family of digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, multichannel buffered serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

**TMS320C6000 Code Composer Studio Tutorial Online Help** (literature number SPRH125) introduces the Code Composer Studio integrated development environment and software tools. Of special interest to DSP/BIOS users are the *Using DSP/BIOS* lessons.

**TMS320C6000 Chip Support Library API Reference Guide** (literature number SPRU401) contains a reference for the Chip Support Library (CSL) application programming interfaces (APIs). The CSL is a set of APIs used to configure and control all on-chip peripherals.

## 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, TMS320C28x, 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.

November 9, 2010

# Contents

---

---

---

<b>1</b>	<b>API Functional Overview</b>	<b>1-1</b>
	<i>This chapter provides an overview to the TMS320C6000 DSP/BIOS API functions.</i>	
1.1	DSP/BIOS Modules	1-2
1.2	Naming Conventions	1-3
1.3	Assembly Language Interface Overview	1-3
1.4	DSP/BIOS Tconf Overview	1-4
1.5	List of Operations	1-5
<b>2</b>	<b>Application Program Interface</b>	<b>2-1</b>
	<i>This chapter describes the DSP/BIOS API modules and functions.</i>	
2.1	ATM Module	2-2
2.2	BCACHE Module	2-15
2.3	BUF Module	2-31
2.4	C62 and C64 Modules	2-42
2.5	CLK Module	2-51
2.6	DEV Module	2-71
2.7	ECM Module	2-118
2.8	GBL Module	2-127
2.9	GIO Module	2-143
2.10	HOOK Module	2-162
2.11	HST Module	2-168
2.12	HWI Module	2-173
2.13	IDL Module	2-202
2.14	LCK Module	2-206
2.15	LOG Module	2-213
2.16	MBX Module	2-230
2.17	MEM Module	2-236
2.18	MPC Module	2-260
2.19	MSGQ Module	2-270
2.20	PIP Module	2-306
2.21	POOL Module	2-326
2.22	PRD Module	2-331
2.23	PWRM Module	2-339
2.24	QUE Module	2-388
2.25	RTDX Module	2-405
2.26	SEM Module	2-421

2.27	SIO Module . . . . .	2-434
2.28	STS Module . . . . .	2-464
2.29	SWI Module. . . . .	2-474
2.30	SYS Module . . . . .	2-503
2.31	TRC Module . . . . .	2-519
2.32	TSK Module . . . . .	2-524
2.33	std.h and stdlib.h functions . . . . .	2-562
<b>A</b>	<b>Function Callability and Error Tables . . . . .</b>	<b>A-1</b>
	<i>This appendix provides tables describing TMS320C6000 errors and function callability.</i>	
A.1	Function Callability Table . . . . .	A-2
A.2	DSP/BIOS Error Codes. . . . .	A-12
<b>B</b>	<b>C6000 DSP/BIOS Register Usage . . . . .</b>	<b>B-1</b>
	<i>This appendix provides tables describing the TMS320C6000TM register conventions in terms of preservation across multi-threaded context switching and preconditions.</i>	
B.1	Overview . . . . .	B-2
B.2	Register Conventions . . . . .	B-2
<b>C</b>	<b>C64x+ Exception Support . . . . .</b>	<b>C-1</b>
	<i>This appendix provides describes support for C64x+ exception handling.</i>	
C.1	C64x+ Exception Support . . . . .	C-2
C.2	Using the DSP/BIOS EXC Module . . . . .	C-3
C.3	Data Types and Macros . . . . .	C-6
C.4	EXC Module . . . . .	C-7
C.5	_MPC Module . . . . .	C-16



# Figures

---

---

---

2-1	Writers and Reader of a Message Queue .....	2-273
2-2	Components of the MSGQ Architecture .....	2-274
2-3	MSGQ Function Calling Sequence .....	2-274
2-4	Pipe Schematic .....	2-308
2-5	Allocators and Message Pools.....	2-327
2-6	Buffer Layout as Defined by STATICPOOL_Params .....	2-329
2-7	PRD Tick Cycles .....	2-336
2-8	Statistics Accumulation on the Host.....	2-467

# Tables

1-1	DSP/BIOS Modules .....	1-2
1-2	DSP/BIOS Operations .....	1-5
2-1	Timer Counter Rates, Targets, and Resets .....	2-53
2-2	High-Resolution Time Determination .....	2-54
2-4	HWI interrupts for the TMS320C6000 .....	2-182
2-5	Conversion Characters for LOG_printf .....	2-224
2-6	Typical Memory Segments for c6x EVM Boards.....	2-248
2-7	Typical Memory Segment for c6711 DSK Boards .....	2-248
2-8	Statistics Units for HWI, PIP, PRD, and SWI Modules .....	2-465
2-9	Conversion Characters Recognized by SYS_printf .....	2-510
2-10	Conversion Characters Recognized by SYS_sprintf .....	2-512
2-11	Conversion Characters Recognized by SYS_vprintf .....	2-514
2-12	Conversion Characters Recognized by SYS_vsprintf .....	2-516
2-13	Events and Statistics Traced by TRC.....	2-519
A-1	Function Callability .....	A-2
A-2	RTS Function Calls .....	A-11
A-3	Error Codes.....	A-12
B-1	Register and Status Bit Handling .....	B-2

# API Functional Overview

---

---

---

This chapter provides an overview to the TMS320C6000 DSP/BIOS API functions.

<b>Topic</b>	<b>Page</b>
<b>1.1 DSP/BIOS Modules . . . . .</b>	<b>1-2</b>
<b>1.2 Naming Conventions. . . . .</b>	<b>1-3</b>
<b>1.3 Assembly Language Interface Overview. . . . .</b>	<b>1-3</b>
<b>1.4 DSP/BIOS Tconf Overview . . . . .</b>	<b>1-4</b>
<b>1.5 List of Operations . . . . .</b>	<b>1-5</b>

## 1.1 DSP/BIOS Modules

Table 1-1. DSP/BIOS Modules

Module	Description
ATM Module	Atomic functions written in assembly language
BCACHE Module	Cache operation manager (C64x+ only)
BUF Module	Maintains buffer pools of fixed size buffers
C62 and C64 Modules	Target-specific functions
CLK Module	System clock manager
DEV Module	Device driver interface
ECM Module	Event combiner manager (C64x+ only)
EXC Module	Exception manager (C64x+ only)
GBL Module	Global setting manager
GIO Module	I/O module used with IOM mini-drivers
HOOK Module	Hook function manager
HST Module	Host channel manager
HWI Module	Hardware interrupt manager
IDL Module	Idle function and processing loop manager
LCK Module	Resource lock manager
LOG Module	Event Log manager
MBX Module	Mailboxes manager
MEM Module	Memory manager
MPC Module	Memory protection manager (C64x+ only)
MSGQ Module	Variable-length message manager
PIP Module	Buffered pipe manager
POOL Module	Allocator interface module
PRD Module	Periodic function manager
PWRM Module	Reduce application's power consumption
QUE Module	Queue manager
RTDX Module	Real-time data exchange manager
SEM Module	Semaphores manager
SIO Module	Stream I/O manager
STS Module	Statistics object manager
SWI Module	Software interrupt manager

Module	Description
SYS Module	System services manager
TRC Module	Trace manager
TSK Module	Multitasking manager
std.h and stdlib.h functions	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.

## 1.3 Assembly Language Interface Overview

The assembly interface that was provided for some of the DSP/BIOS APIs has been deprecated. They are no longer documented.

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 *TMS320C6000 Optimizing Compiler User's Guide* for more details.

When you are using the DSP/BIOS Configuration Tool, use a leading underscore before the name of any C function you configure. (The DSP/BIOS Configuration Tool generates assembly code, but does not add the underscore automatically.) If you are using Tconf, do not add an underscore before the function name; Tconf internally adds the underscore needed to call a C function from assembly.

All DSP/BIOS APIs follow standard C calling conventions as documented in the C programmer's guide for the device you are using.

DSP/BIOS APIs save and restore context for each thread during a context switch. Your code should simply follow standard C register usage conventions. Code written in assembly language should be written to conform to the register usage model specified in the C compiler manual for your device. When writing assembly language, take special care to make sure the C context is preserved. For example, if you change the AMR register on the 'C6000, you should be sure to change it back before returning from your assembly language routine. See the Register Usage appendix in this book to see how DSP/BIOS uses specific registers.

## 1.4 DSP/BIOS Tconf Overview

The section describing each modules in this manual lists properties that can be configured in Tconf scripts, along with their types and default values. The sections on manager properties and instance properties also provide Tconf examples that set each property.

For details on Tconf scripts, see the *DSP/BIOS Tconf User's Guide* (SPRU007). The language used is JavaScript with an object model specific to the needs of DSP/BIOS configuration.

In general, property names of Module objects are in all uppercase letters. For example, "STACKSIZE". Property names of Instance objects begin with a lowercase word. Subsequent words have their first letter capitalized. For example, "stackSize".

Default values for many properties are dependent on the values of other properties. The defaults shown are those that apply if related property values have not been modified. The defaults shown are for 'C62x and 'C67x. Memory segment defaults are different for 'C64x. Default values for many HWI properties are different for each instance.

The data types shown for the properties are not used as syntax in Tconf scripts. However, they do indicate the type of values that are valid for each property. The types used are as follows:

- ❑ **Arg.** Arg properties hold arguments to pass to program functions. They may be strings, integers, labels, or other types as needed by the program function.
- ❑ **Bool.** You may assign a value of either true or 1 to set a Boolean property to true. You may assign a value of either false or 0 (zero) to set a Boolean property to false. Do not set a Boolean property to the quoted string "true" or "false".
- ❑ **EnumInt.** Enumerated integer properties accept a set of valid integer values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- ❑ **EnumString.** Enumerated string properties accept certain string values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- ❑ **Extern.** Properties that hold function names use the Extern type. In order to specify a function Extern, use the prog.extern() method as shown in the examples to refer to objects defined as asm, C, or C++ language symbols. The default language is C.
- ❑ **Int16.** Integer properties hold 16-bit unsigned integer values. The value range accepted for a property may have additional limits.

- ❑ **Int32.** Long integer properties hold 32-bit unsigned integer values. The value range accepted for a property may have additional limits.
- ❑ **Numeric.** Numeric properties hold either 32-bit signed or unsigned values or decimal values, as appropriate for the property.
- ❑ **Reference.** Properties that reference other configured objects contain an object reference. Use the `prog.get()` method to specify a reference to another object.
- ❑ **String.** String properties hold text strings.

## 1.5 List of Operations

Table 1-2. DSP/BIOS Operations

### ATM module operations

Function	Operation
ATM_andi, ATM_andu	Atomically AND memory location with mask and return previous value
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 with mask and return previous value
ATM_seti, ATM_setu	Atomically set memory and return previous value

### BCACHE module operations (C64x+ only)

Function	Operation
BCACHE_getMar	Get a MAR register value
BCACHE_getMode	Get L1D, L1P, or L2 cache operating mode
BCACHE_getSize	Get the L1D, L1P, and L2 cache sizes
BCACHE_inv	Invalidate the specified memory range in caches
BCACHE_invL1pAll	Invalidates all lines in L1P cache
BCACHE_setMar	Set a MAR register value
BCACHE_setMode	Set L1D, L1P, or L2 cache operating mode

Function	Operation
BCACHE_setSize	Set the L1D, L1P, and L2 cache sizes
BCACHE_wait	Waits for a previous cache operation to complete
BCACHE_wb	Writes back a range of memory from caches
BCACHE_wbAll	Performs a global write back from caches
BCACHE_wbInv	Writes back and invalidates a range of memory
BCACHE_wbInvAll	Performs a global write back and invalidate

### *BUF module operations*

Function	Operation
BUF_alloc	Allocate a fixed memory buffer out of the buffer pool
BUF_create	Dynamically create a buffer pool
BUF_delete	Delete a dynamically created buffer pool
BUF_free	Free a fixed memory buffer into the buffer pool
BUF_maxbuff	Check the maximum number of buffers used from the buffer pool
BUF_stat	Determine the status of a buffer pool (buffer size, number of free buffers, total number of buffers in the pool)

### *C62/64 operations*

Function	Operation
C62_disableIER, C64_disableIER	Disable certain maskable interrupts
C62_enableIER, C64_enableIER	Enable certain maskable interrupts
C62_plug, C64_plug	C function to plug an interrupt vector



*CLK module operations*

Function	Operation
CLK_countspms	Number of hardware timer counts per millisecond
CLK_cpuCyclesPerHtime	Return multiplier for converting high-res time to CPU cycles
CLK_cpuCyclesPerLtime	Return multiplier for converting low-res time to CPU cycles
CLK_gethtime	Get high-resolution time
CLK_getltime	Get low-resolution time
CLK_getprd	Get period register value
CLK_reconfig	Reset timer period and registers
CLK_start	Restart the low-resolution timer
CLK_stop	Halt the low-resolution timer

*DEV module operations*

Function	Operation
DEV_createDevice	Dynamically creates device with user-defined parameters
DEV_deleteDevice	Deletes the dynamically created device
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
DGN Driver	Software generator driver
DGS Driver	Stackable gather/scatter driver
DHL Driver	Host link driver

Function	Operation
DIO Driver	Class driver
DNL Driver	Null driver
DOV Driver	Stackable overlap driver
DPI Driver	Pipe driver
DST Driver	Stackable split driver
DTR Driver	Stackable streaming transformer driver

*ECM module operations (C64x+ only)*

Function	Operation
ECM_disableEvent	Disable a system event in its event combiner mask
ECM_dispatch	Run functions for a combined event
ECM_dispatchPlug	Specify function and attributes for a system event
ECM_enableEvent	Enable a system event in its event combiner mask

*EXC module operations (C64x+ only)*

Function	Operation
EXC_clearLastStatus	Clears latest exception status values
EXC_dispatch	Function run by HWI_NMI to process exceptions
EXC_evtEvtClear	Clears an event from the event flag register
EXC_evtExpEnable	Enables an event type to an exception
EXC_exceptionHandler	Serves non-software exceptions
EXC_exceptionHook	Hook fxn called by EXC_exceptionHandler
EXC_external	Handles exceptions external to the CPU
EXC_externalHook	Hook fxn called by EXC_external
EXC_getLastStatus	Gets latest exception status values

Function	Operation
EXC_internal	Handles exceptions internal to the CPU
EXC_internalHook	Hook fxn called by EXC_internal
EXC_nmi	Handles legacy NMI exceptions
EXC_nmiHook	Hook fxn called by EXC_nmi

### *GBL module operations*

Function	Operation
GBL_getClkin	Get configured value of board input clock in KHz
GBL_getFrequency	Get current frequency of the CPU in KHz
GBL_getProclId	Get configured processor ID used by MSGQ
GBL_getVersion	Get DSP/BIOS version information
GBL_setFrequency	Set frequency of CPU in KHz for DSP/BIOS
GBL_setProclId	Set configured value of processor ID used by MSGQ

### *GIO module operations*

Function	Operation
GIO_abort	Abort all pending input and output
GIO_control	Device-specific control call
GIO_create	Allocate and initialize a GIO object
GIO_delete	Delete underlying IOM mini-drivers and free GIO object and its structure
GIO_flush	Drain output buffers and discard any pending input
GIO_new	Initialize a pre-allocated GIO object
GIO_read	Synchronous read command
GIO_submit	Submit a GIO packet to the mini-driver
GIO_write	Synchronous write command

### *HOOK module operations*

Function	Operation
HOOK_getenv	Get environment pointer for a given HOOK and TSK combination
HOOK_setenv	Set environment pointer for a given HOOK and TSK combination

### *HST module operations*

Function	Operation
HST_getpipe	Get corresponding pipe object

### *HWI module operations*

Function	Operation
HWI_applyWugenMasks	Apply specified masks to WUGEN interrupt mask registers
HWI_disable	Globally disable hardware interrupts
HWI_disableWugen	Disable an event in the WUGEN interrupt mask registers
HWI_dispatchPlug	Plug the HWI dispatcher
HWI_enable	Globally enable hardware interrupts
HWI_enableWugen	Enable an event in the WUGEN interrupt mask registers
HWI_enter	Hardware interrupt service routine prolog
HWI_eventMap	Assign interrupt selection number to HWI object
HWI_exit	Hardware interrupt service routine epilog
HWI_getWugenMasks	Get masks from WUGEN interrupt mask registers
HWI_iierToWugenMasks	Compute WUGEN masks from IER register
HWI_isHWI	Check to see if called in the context of an HWI
HWI_restore	Restore global interrupt enable state

### *IDL module operations*

Function	Operation
IDL_run	Make one pass through idle functions

*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

*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_event5	Append a 5-argument unformatted message to a log
LOG_printf	Append a formatted message to a message log
LOG_printf4	Append a 4-argument formatted message to a message log
LOG_reset	Reset a log

*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

*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_getBaseAddress	Get base address of a memory heap
MEM_increaseTableSize	Increase the internal MEM table size
MEM_redefine	Redefine an existing memory heap
MEM_stat	Return the status of a memory heap
MEM_undefine	Undefine an existing memory segment

*MPC module operations (C64x+ only)*

Function	Operation
MPC_getPA	Get permission attributes of address
MPC_getPageSize	Get size of page containing address
MPC_getPrivMode	Get current CPU privilege mode
MPC_setBufferPA	Set permission attributes for a buffer
MPC_setPA	Set permission attributes for an address
MPC_setPrivMode	Set CPU privilege mode

*\_MPC module operations (C64x+ only)*

Function	Operation
_MPC_getLastMPFAR	Gets MPFAR for a memory controller
_MPC_getLastMPFSR	Gets MPFSR for a memory controller

Function	Operation
_MPC_exceptionHandler	Assigned to EXC_exceptionHook.
_MPC_externalHandler	Assigned to EXC_externalHook
_MPC_internalHandler	Assigned to EXC_internalHook
_MPC_userHook	Hook for user-defined function

### *MSGQ module operations*

Function	Operation
MSGQ_alloc	Allocate a message. Performed by writer.
MSGQ_close	Closes a message queue. Performed by reader.
MSGQ_count	Return the number of messages in a message queue
MSGQ_free	Free a message. Performed by reader.
MSGQ_get	Receive a message from the message queue. Performed by reader.
MSGQ_getAttrs	Get attributes of a message queue.
MSGQ_getDstQueue	Get destination message queue field in a message.
MSGQ_getMsgId	Return the message ID from a message.
MSGQ_getMsgSize	Return the message size from a message.
MSGQ_getSrcQueue	Extract the reply destination from a message.
MSGQ_isLocalQueue	Return whether queue is local.
MSGQ_locate	Synchronously find a message queue. Performed by writer.
MSGQ_locateAsync	Asynchronously find a message queue. Performed by writer.
MSGQ_open	Opens a message queue. Performed by reader.
MSGQ_put	Place a message on a message queue. Performed by writer.
MSGQ_release	Release a located message queue. Performed by writer.
MSGQ_setErrorHandler	Set up handling of internal MSGQ errors.
MSGQ_setMsgId	Sets the message ID in a message.
MSGQ_setSrcQueue	Sets the reply destination in a message.

*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

*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

*PWRM module operations ('C6748 devices)*

Function	Operation
PWRM_changeSetpoint	Initiate a change to the V/F setpoint
PWRM_configure	Set new configuration parameters for PWRM
PWRM_getCapabilities	Get information on PWRM's capabilities on the current platform
PWRM_getConstraintInfo	Get information on constraints registered with PWRM
PWRM_getCPULoad	Get CPU load information



Function	Operation
PWRM_getCurrentSetpoint	Get the current setpoint in effect
PWRM_getDependencyCount	Get count of dependencies currently declared on a resource
PWRM_getLoadMonitorInfo	Get PWRM load monitor configuration
PWRM_getNumSetpoints	Get the number of setpoints supported for the current platform
PWRM_getSetpointInfo	Get the corresponding frequency and CPU core voltage for a setpoint
PWRM_getTransitionLatency	Get the latency to scale between setpoints
PWRM_idleClocks	Immediately idle the clock domains
PWRM_registerConstraint	Register an operational constraint with PWRM
PWRM_registerNotify	Register a function to be called on a specific power event
pwrM_notifyFxn	Function to be called on a registered power event
PWRM_releaseDependency	Release a dependency that has been previously declared
PWRM_resetCPULoadHistory	Clear the CPU load history buffered by PWRM
PWRM_setDependency	Declare a dependency upon a resource
PWRM_signalEvent	Signal a power event to registered notification clients
PWRM_sleepDSP	Transition the DSP to a new sleep state
PWRM_startCPULoadMonitoring	Restart CPU load monitoring
PWRM_stopCPULoadMonitoring	Stop CPU load monitoring
PWRM_unregisterConstraint	Unregister a constraint from PWRM
PWRM_unregisterNotify	Unregister for an event notification from PWRM

### *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)

<b>Function</b>	<b>Operation</b>
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)

*RTDX module operations*

<b>Function</b>	<b>Operation</b>
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

*SEM module operations*

<b>Function</b>	<b>Operation</b>
SEM_count	Get current semaphore count
SEM_create	Create a semaphore
SEM_delete	Delete a semaphore
SEM_new	Initialize a semaphore
SEM_pend	Wait for a counting semaphore
SEM_pendBinary	Wait for a binary semaphore
SEM_post	Signal a counting semaphore
SEM_postBinary	Signal a binary semaphore
SEM_reset	Reset semaphore

*SIO module operations*

<b>Function</b>	<b>Operation</b>
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_ready	Determine if device for stream is ready
SIO_reclaim	Request a buffer back from a stream
SIO_reclaimx	Request a buffer and frame status back from a stream

Function	Operation
SIO_segid	Memory section used by a stream
SIO_select	Select a ready device
SIO_staticbuf	Acquire static buffer from stream

### *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

### *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_getmbox	Return SWI's mailbox value
SWI_getpri	Return an SWI's priority mask
SWI_inc	Increment SWI's mailbox and post
SWI_isSWI	Check to see if called in the context of a SWI
SWI_or	Set or mask in an SWI's mailbox and post
SWI_orHook	Specialized version of SWI_or
SWI_post	Post a software interrupt
SWI_raisepri	Raise an SWI's priority

Function	Operation
SWI_restorepri	Restore an SWI's priority
SWI_self	Return address of currently executing SWI object
SWI_setattr	Set attributes of a software interrupt

### *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

### *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

### *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

Function	Operation
TSK_getname	Get task name
TSK_getpri	Get task priority
TSK_getsts	Get task STS object
TSK_isTSK	Check to see if called in the context of a TSK
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_time	Return current value of system clock
TSK_yield	Yield processor to equal priority task

### *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

### *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 DSP/BIOS API modules and functions.

Topic	Page
2.1 ATM Module . . . . .	2-2
2.2 BCACHE Module . . . . .	2-15
2.3 BUF Module . . . . .	2-31
2.4 C62 and C64 Modules . . . . .	2-42
2.5 CLK Module . . . . .	2-51
2.6 DEV Module . . . . .	2-71
2.7 ECM Module . . . . .	2-118
2.8 GBL Module . . . . .	2-127
2.9 GIO Module. . . . .	2-143
2.10 HOOK Module . . . . .	2-162
2.11 HST Module . . . . .	2-168
2.12 HWI Module . . . . .	2-173
2.13 IDL Module . . . . .	2-202
2.14 LCK Module . . . . .	2-206
2.15 LOG Module . . . . .	2-213
2.16 MBX Module . . . . .	2-230
2.17 MEM Module. . . . .	2-236
2.18 MPC Module . . . . .	2-260
2.19 MSGQ Module . . . . .	2-270
2.20 PIP Module . . . . .	2-306
2.21 POOL Module. . . . .	2-326
2.22 PRD Module . . . . .	2-331
2.23 PWRM Module . . . . .	2-339
2.24 QUE Module . . . . .	2-388
2.25 RTDX Module . . . . .	2-405
2.26 SEM Module . . . . .	2-421
2.27 SIO Module. . . . .	2-434
2.28 STS Module . . . . .	2-464
2.29 SWI Module . . . . .	2-474
2.30 SYS Module . . . . .	2-503
2.31 TRC Module . . . . .	2-519
2.32 TSK Module . . . . .	2-524
2.33 std.h and stdlib.h functions . . . . .	2-562

## 2.1 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**

<b>Syntax</b>	ival = ATM_andi(idst, isrc);		
<b>Parameters</b>	volatile Int	*idst;	/* pointer to integer */
	Int	isrc;	/* integer mask */
<b>Return Value</b>	Int	ival;	/* previous value of *idst */

**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**

<code>volatile Uns</code>	<code>*udst;</code>	<code>/* pointer to unsigned */</code>
<code>Uns</code>	<code>usrc;</code>	<code>/* unsigned mask */</code>

**Return Value**

<code>Uns</code>	<code>uval;</code>	<code>/* previous value of *udst */</code>
------------------	--------------------	--

**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   *dst;     /* pointer to integer */</code>
<b>Return Value</b>	<code>Int            ival;     /* previous value of *dst */</code>

**Description**

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

```
`interrupt disable`  
ival = *dst;  
*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 */`**Description**

ATM\_clearu atomically clears an Uns memory location and returns its previous value as follows:

```
`interrupt disable`  
uval = *udst;  
*udst = 0;  
`interrupt enable`  
return (uval);
```

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

**See Also**

ATM\_cleari  
ATM\_setu

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

**ATM\_decu***Atomically decrement Uns memory and return new value***C Interface****Syntax**`uval = ATM_decu(udst);`**Parameters**`volatile Uns *udst; /* pointer to unsigned */`**Return Value**`Uns uval; /* new value after decrement */`**Description**

ATM\_decu atomically decrements a Uns memory location and returns its new value as follows:

```
`interrupt disable`  
uval = *udst - 1;  
*udst = uval;  
`interrupt enable`  
return (uval);
```

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

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

**See Also**

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 */`**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

**ATM\_incu***Atomically increment Uns memory and return new value***C Interface****Syntax**`uval = ATM_incu(udst);`**Parameters**`volatile Uns *udst; /* pointer to unsigned */`**Return Value**`Uns uval; /* new value after increment */`**Description**

ATM\_incu atomically increments an Uns memory location and returns its new value as follows:

```
`interrupt disable`  
uval = *udst + 1;  
*udst = uval;  
`interrupt enable`  
return (uval);
```

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

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

**See Also**

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>
------------------	--------------------	--

**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 */
```

**Description**

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:

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

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

**See Also**

```
ATM_andu  
ATM_ori
```

**ATM\_seti***Atomically set Int memory and return previous value***C Interface**

<b>Syntax</b>	<code>iold = ATM_seti(idst, inew);</code>
<b>Parameters</b>	<code>volatile Int   *idst;    /* pointer to integer */</code> <code>Int            inew;    /* new integer value */</code>
<b>Return Value</b>	<code>Int            iold;    /* previous value of *idst */</code>

**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 */`**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`

## 2.2 BCACHE Module

The BCACHE module provides DSP/BIOS support for the C64x+ L1/L2 cache. This module is available only for C64x+ devices.

### Functions

- ❑ BCACHE\_getMar. Get a MAR register value.
- ❑ BCACHE\_getMode. Get L1D, L1P, or L2 cache operating mode
- ❑ BCACHE\_getSize. Get the L1D, L1P, and L2 cache sizes
- ❑ BCACHE\_inv. Invalidate the specified memory range in caches
- ❑ BCACHE\_invL1pAll. Invalidates all lines in L1P cache
- ❑ BCACHE\_setMar. Set a MAR register value
- ❑ BCACHE\_setMode. Set L1D, L1P, or L2 cache operating mode
- ❑ BCACHE\_setSize. Set the L1D, L1P, and L2 cache sizes
- ❑ BCACHE\_wait. Waits for a previous cache operation to complete
- ❑ BCACHE\_wb. Writes back a range of memory from caches
- ❑ BCACHE\_wbAll. Performs a global write back from caches
- ❑ BCACHE\_wbInv. Writes back and invalidates a range of memory
- ❑ BCACHE\_wbInvAll. Performs a global write back and invalidate

### Constants, Types, and Structures

```
/* Enumerated list of L1 cache sizes */
typedef enum {
    BCACHE_L1_0K = 0,
    BCACHE_L1_4K = 1,
    BCACHE_L1_8K = 2,
    BCACHE_L1_16K = 3,
    BCACHE_L1_32K = 4
} BCACHE_L1_Size;

/* Enumerated list of L2 cache sizes */
typedef enum {
    BCACHE_L2_0K = 0,
    BCACHE_L2_32K = 1,
    BCACHE_L2_64K = 2,
    BCACHE_L2_128K = 3,
    BCACHE_L2_256K = 4
} BCACHE_L2_Size;

/* Enumerated list of cache modes for L1/L2 caches */
typedef enum {
    BCACHE_NORMAL,
    BCACHE_FREEZE,
    BCACHE_BYPASS
} BCACHE_Mode;
```

```
/* Enumerated list of caches */
typedef enum {
    BCACHE_L1D,
    BCACHE_L1P,
    BCACHE_L2
} BCACHE_Level;

/* Enumerated list of MAR values */
typedef enum {
    BCACHE_MAR_DISABLE = 0,
    BCACHE_MAR_ENABLE = 1,
} BCACHE_Mar;

/* L1 and L2 cache size structure */
typedef struct BCACHE_Size {
    BCACHE_L1_Size l1psize;
    BCACHE_L1_Size l1dsize;
    BCACHE_L2_Size l2size;
} BCACHE_Size;
```

## Description

The BCACHE module supports the C64x+ caches. The caches on these devices are Level 1 Program (L1P), Level 1 Data (L1D), and Level 2 (L2). See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about the L1P, L1D, and L2 caches.

This module provides API functions that perform cache coherency operations at the cache line level or globally. The cache coherency operations are:

- ❑ **Invalidate.** Makes valid cache lines invalid and discards the content of the affected cache lines.
- ❑ **Writeback.** Writes the contents of cache lines to a lower-level memory, such as the L2 cache and/or external memory, without discarding the lines in the original cache.
- ❑ **Writeback-Invalidation.** Writes the contents of cache lines to lower-level memory, and then discards the contents of the lines in the original cache.

This module also provides API functions that get and set the size and mode of the caches. You can also get and set registers that indicate whether a particular memory range is cacheable.

This module has no configuration interface.

**BCACHE\_getMar***Get a MAR register value***C Interface**

**Syntax** `marVal = BCACHE_getMar(baseAddr)`

**Parameters** `Ptr baseAddr; /* address of memory range */`

**Return Value** `BCACHE_Mar marVal; /* value of specified MAR register */`

**Description**

This function is available only for C64x+ devices.

BCACHE\_getMar gets the value of the specified MAR register.

The C64x+ L2 memory includes a set of registers that define the cacheability of external memory ranges. The registers, referred to as MARs (Memory Attribute Registers), are defined as shown in Table 4-33.

For baseAddr parameter, specify the base address of the memory range for which you want to know the cacheability of the memory. Do not use the base address of the MAR register itself.

This function returns the value of the MAR bit that indicates whether the corresponding memory range is cacheable. The value is 0 for non-cacheable memory and 1 for cacheable memory. The BCACHE\_Mar type provides the following constants for testing this bit:

```
/* Enumerated list of MAR values */
typedef enum {
    BCACHE_MAR_DISABLE = 0,
    BCACHE_MAR_ENABLE = 1,
} BCACHE_Mar;
```

You can use the BCACHE\_setMar function to set the value of a MAR.

**Constraints and Calling Context**

❑ none

**See Also**

BCACHE\_setMar

**BCACHE\_getMode***Get L1D, L1P, or L2 cache operating mode***C Interface**

**Syntax** `cacheMode = BCACHE_getMode(level)`

**Parameters** `BCACHE_Level level; /* cache to use */`

**Return Value** `BCACHE_Mode cacheMode; /* current mode */`

**Description**

This function is available only for C64x+ devices.

`BCACHE_getMode` gets the cache operating mode for the specified L1D, L1P, or L2 cache.

The level parameter specifies which cache to use. The `BCACHE_Level` type provides the following constants for specifying a cache:

```
/* Enumerated list of caches */
typedef enum {
    BCACHE_L1D,
    BCACHE_L1P,
    BCACHE_L2
} BCACHE_Level;
```

This function returns the current cache mode for the specified cache. See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about cache modes. The `BCACHE_Mode` type provides the following constants for cache modes:

```
/* Enumerated list of cache modes for L1/L2 caches */
typedef enum {
    BCACHE_NORMAL,
    BCACHE_FREEZE,
    BCACHE_BYPASS
} BCACHE_Mode;
```

Freeze mode is supported for all caches. Bypass mode is supported only for the L2 cache.

You can use the `BCACHE_setMode` function to set the mode of a cache.

**Constraints and Calling Context**

❑ none

**See Also**

`BCACHE_setMode`



## **BCACHE\_getSize** *Get the L1D, L1P, and L2 cache sizes*

### **C Interface**

<b>Syntax</b>	BCACHE_getSize(*size)
<b>Parameters</b>	BCACHE_Size *size; /* sizes of caches */
<b>Return Value</b>	Void

### **Description**

This function is available only for C64x+ devices.

BCACHE\_getSize gets the size of the L1D, L1P, and L2 caches.

The size parameter is a pointer to a structure that returns the size of the caches. The structure is defined as follows:

```
/* L1 and L2 cache size structure */
typedef struct BCACHE_Size {
    BCACHE_L1_Size l1psize;
    BCACHE_L1_Size l1dsize;
    BCACHE_L2_Size l2size;
} BCACHE_Size;
```

You can use the BCACHE\_setSize function to set the cache sizes.

### **Constraints and Calling Context**

❑ none

### **See Also**

BCACHE\_setSize

**BCACHE\_inv***Invalidate the specified memory range in caches***C Interface****Syntax**

BCACHE\_inv(blockPtr, byteCnt, wait)

**Parameters**

Ptr	blockPtr;	/* start address of range to be invalidated */
size_t	byteCnt;	/* number of bytes to be invalidated */
Bool	wait;	/* wait until the operation is completed */

**Return Value**

Void

**Description**

This function is available only for C64x+ devices.

BCACHE\_inv invalidates a range of memory from all caches. When you invalidate a cache line, its contents are discarded and the cache tags the line as "dirty" so that next time that particular address is read, it is obtained from external memory.

The blockPtr points to an address in non-cache memory that may be cached in L1P, L1D, L2, or not at all. If the blockPtr does not correspond to the start of a cache line, the start of that cache line is used.

If the byteCnt is not equal to a whole number of cache lines, the byteCnt is rounded up to the next size that equals a whole number of cache lines. L1P cache lines are 32 bytes. L1D cache lines are 64 bytes. L2 cache lines are 128 bytes.

If the wait parameter is true, then this function waits until the invalidation operation is complete to return. If the wait parameter is false, this function returns immediately. You can use BCACHE\_wait later to ensure that this operation is complete.

This function always waits for other cache operations to finish before performing its actions.

All lines in the specified range are invalidated in any cache location where that address is cached. See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for more on cache line invalidation.

**Constraints and Calling Context**☐ none**See Also**

BCACHE\_invL1pAll  
BCACHE\_wait  
BCACHE\_wbInv  
BCACHE\_wbInvAll

**BCACHE\_invL1pAll***Invalidates all lines in L1P cache***C Interface****Syntax** BCACHE\_invL1pAll()**Parameters** Void**Return Value** Void**Description**

This function is available only for C64x+ devices.

BCACHE\_invL1pAll invalidates the L1P cache completely. This discards the entire contents of the L1P cache.

This function always waits for other cache operations to finish before performing its actions. This function always waits until its invalidation operation is complete to return.

See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for more on cache invalidation.

**Constraints and  
Calling Context**

□ none

**See Also**

BCACHE\_inv  
BCACHE\_wbInv  
BCACHE\_wbInvAll

**BCACHE\_setMar***Set a MAR register value***C Interface**

<b>Syntax</b>	BCACHE_setMar(baseAddr, byteSize, value)		
<b>Parameters</b>	Ptr	baseAddr;	/* base address of the range */
	size_t	byteSize;	/* size in bytes used to determine range */
	BCACHE_MAR	value;	/* the value to which MARs should be set */
<b>Return Value</b>	Void		

**Description**

This function is available only for C64x+ devices.

BCACHE\_setMar sets the value of the specified MAR register or registers.

The C64x+ L2 memory includes a set of registers that define the cacheability of external memory ranges. The registers, referred to as MARs (Memory Attribute Registers), are defined as shown in Table 4-33.

For baseAddr parameter, specify the base address of the memory range for which you want to set the cacheability of the memory. Do not use the base address of the MAR register itself.

The byteSize allows you to specify the size of the memory range. Together, the baseAddr and byteSize are use to determine the number of MAR registers to set. For example, suppose you have the following values:

```
baseAddr = 0x80000000
byteSize = 0x10000000
```

This would mean BCACHE\_setMar should set MAR128 through MAR144 because:

```
MAR128 corresponds to 0x80000000-0x80FFFFFF
MAR129 corresponds to 0x81000000-0x81FFFFFF
. . .
MAR144 corresponds to 0x90000000-0x90FFFFFF
```

However if byteSize were 0x00001000, this function would set only MAR128.

This function sets the value of the MAR bit that indicates whether the corresponding memory range is cacheable. The value is 0 for non-cacheable memory and 1 for cacheable memory. The BCACHE\_Mar type provides the following constants for setting this bit:

```
/* Enumerated list of MAR values */
typedef enum {
    BCACHE_MAR_DISABLE = 0,
    BCACHE_MAR_ENABLE = 1,
} BCACHE_Mar;
```

You can use the BCACHE\_getMar function to get the value of a MAR.

**Constraints and  
Calling Context**

❑ none

**See Also**

BCACHE\_getMar

**BCACHE\_setMode***Set L1D, L1P, or L2 cache operating mode***C Interface**

**Syntax** `oldCacheMode = BCACHE_setMode(level, newCacheMode)`

**Parameters** `BCACHE_Level level; /* cache to use */`  
`BCACHE_Mode newCacheMode; /* the new mode to be applied */`

**Return Value** `BCACHE_Mode oldCacheMode; /* the previous mode */`

**Description**

This function is available only for C64x+ devices.

BCACHE\_setMode sets the cache operating mode for the specified L1D, L1P, or L2 cache.

The level parameter specifies which cache to set. The BCACHE\_Level type provides the following constants for specifying a cache:

```
/* Enumerated list of caches */
typedef enum {
    BCACHE_L1D,
    BCACHE_L1P,
    BCACHE_L2
} BCACHE_Level;
```

The newCacheMode parameter indicates the new mode to use for the specified cache. See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about cache modes. The BCACHE\_Mode type provides the following constants for cache modes:

```
/* Enumerated list of cache modes for L1/L2 caches */
typedef enum {
    BCACHE_NORMAL,
    BCACHE_FREEZE,
    BCACHE_BYPASS
} BCACHE_Mode;
```

Freeze mode is supported for all caches. Bypass mode is supported only for the L2 cache.

This function returns the previous cache mode using the same constants as for the newCacheMode parameter.

You can use the BCACHE\_getMode function to get the mode of a cache.

**Constraints and Calling Context**

❑ none

**See Also**

BCACHE\_getMode

## BCACHE\_setSize *Set the L1D, L1P, and L2 cache sizes*

### C Interface

<b>Syntax</b>	BCACHE_setSize(*size)
<b>Parameters</b>	BCACHE_Size *size; /* sizes of caches */
<b>Return Value</b>	Void

### Description

This function is available only for C64x+ devices.

BCACHE\_setSize sets the size of the L1D, L1P, and L2 caches.

The size parameter is a pointer to a structure that specifies the new sizes of the caches. The structure is defined as follows:

```
/* L1 and L2 cache size structure */
typedef struct BCACHE_Size {
    BCACHE_L1_Size l1psize;
    BCACHE_L1_Size l1dsize;
    BCACHE_L2_Size l2size;
} BCACHE_Size;
```

The size of the L1D and L1P cache may be 0 KB, 4 KB, 8 KB, 16 KB, or 32 KB. The size of the L2 cache may be 0 KB, 32 KB, 64 KB, 128 KB, or 256 KB.

When you change the L1D or L2 cache size, that cache writes-back and invalidates its current contents. When you change the L1P cache size, the L1P cache invalidates its current contents. See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about data loss issues when changing the cache size.

You can use the BCACHE\_getSize function to get the cache sizes.

### Constraints and Calling Context

❑ none

### See Also

BCACHE\_getSize

**BCACHE\_wait***Waits for a previous cache operation to complete***C Interface****Syntax** BCACHE\_wait()**Parameters** Void**Return Value** Void**Description**

This function is available only for C64x+ devices.

BCACHE\_wait will wait for a previously issued cache operation (invalidate, writeback, or writeback with invalidate) to complete. If no cache operation is pending, the function simply returns.

You can use this function if you set the wait parameter to false in the previous BCACHE function and, at some later point, want to make sure that operation is complete.

The BCACHE APIs use this function internally check to see if all cache operations are complete before performing their own operations. Thus, you would only need to use this function before statements that may affect a cache but that do not use the BCACHE APIs.

For the OMAP 2430/3430, BCACHE\_wait reads the addresses specified by the GBL.BCACHEREADADDR0 to GBL.BCACHEREADADDR2 parameters described in the GBL Module Properties section. Reading a non-cached address is necessary to ensure that the writeback has fully completed.

You do not need to call this function for global cache operations such as BCACHE\_wbAll. Those functions always wait for the operation to finish before returning.

**Constraints and Calling Context**

❑ none

**See Also**

BCACHE\_inv  
BCACHE\_wb  
BCACHE\_wbInv



**BCACHE\_wb***Writes back a range of memory from caches***C Interface****Syntax**

BCACHE\_wb(blockPtr, byteCnt, wait)

**Parameters**

Ptr	blockPtr;	/* start address of range to writeback */
size_t	byteCnt;	/* number of bytes to writeback */
Bool	wait;	/* wait until the operation is completed */

**Return Value**

Void

**Description**

This function is available only for C64x+ devices.

BCACHE\_wb writes back the range of memory from all caches that can be written back. When you perform a writeback, the contents of the cache lines are written to lower-level memory.

The blockPtr points to an address in non-cache memory that may be cached in L1P, L1D, L2, or not at all. If the blockPtr does not correspond to the start of a cache line, the start of that cache line is used.

If the byteCnt is not equal to a whole number of cache lines, the byteCnt is rounded up to the next size that equals a whole number of cache lines. L1D cache lines are 64 bytes. L2 cache lines are 128 bytes.

If the wait parameter is true, then this function waits until the write back operation is complete to return. If the wait parameter is false, this function returns immediately. You can use BCACHE\_wait later to ensure that this operation is complete.

- ❑ In L1P no changes are made.
- ❑ In L1D all lines in the range are left valid in the L1D cache and data in the range is written back to L2 and/or external memory.
- ❑ In L2 all lines in the range are left valid in the L2 cache and data in the range is written back to external memory.

This function always waits for other cache operations to finish before performing its actions.

**Constraints and Calling Context**

- ❑ none

**See Also**

BCACHE\_inv  
 BCACHE\_wbAll  
 BCACHE\_wbInv  
 BCACHE\_wbInvAll

**BCACHE\_wbAll***Performs a global write back from caches***C Interface****Syntax** BCACHE\_wbAll()**Parameters** Void**Return Value** Void**Description**

This function is available only for C64x+ devices.

BCACHE\_wbAll performs a global writeback.

- ☐ There is no effect on the L1P cache.
- ☐ All lines are left valid in the L1D cache and the data in the L1D cache is written back to L2 or external.
- ☐ All lines are left valid in the L2 cache and the data in the L2 cache is written back to external.

This function always waits for other cache operations to finish before performing its actions. This function always waits until its writeback operation is complete to return.

See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for more on cache writebacks.

**Constraints and  
Calling Context**

- ☐ none

**See Also**

BCACHE\_wb  
BCACHE\_wbInv  
BCACHE\_wbInvAll

**BCACHE\_wbInv***Writes back and invalidates a range of memory***C Interface****Syntax**

BCACHE\_wbInv(blockPtr, byteCnt, wait)

**Parameters**

Ptr	blockPtr;	/* start address of range to writeback/inv */
size_t	byteCnt;	/* number of bytes to writeback/invalidate */
Bool	wait;	/* wait until the operation is completed */

**Return Value**

Void

**Description**

This function is available only for C64x+ devices.

BCACHE\_wbInv writes back and invalidates the range of memory in all caches. When you perform a writeback, the contents of the cache lines are written to lower-level memory. When you invalidate a cache line, its contents are discarded.

The blockPtr points to an address in non-cache memory that may be cached in L1P, L1D, L2, or not at all. If the blockPtr does not correspond to the start of a cache line, the start of that cache line is used.

If the byteCnt is not equal to a whole number of cache lines, the byteCnt is rounded up to the next size that equals a whole number of cache lines. L1P cache lines are 32 bytes. L1D cache lines are 64 bytes. L2 cache lines are 128 bytes.

If the wait parameter is true, then this function waits until the writeback and invalidate operation is complete to return. If the wait parameter is false, this function returns immediately. You can use BCACHE\_wait later to ensure that this operation is complete.

- ☐ In L1P all lines in the range are invalidated but not written back.
- ☐ In L1D all lines in the range are invalidated in the L1D cache and data in the range is written back to L2 and/or external memory.
- ☐ In L2 all lines in the range are invalidated in the L2 cache and data in the range is written back to external memory.

This function always waits for other cache operations to finish before performing its actions.

**Constraints and Calling Context**

- ☐ none

**See Also**

BCACHE\_inv  
 BCACHE\_wb  
 BCACHE\_wbInvAll

**BCACHE\_wbInvAll** *Performs a global write back and invalidate***C Interface****Syntax** BCACHE\_wbInvAll()**Parameters** Void**Return Value** Void**Description**

This function is available only for C64x+ devices.

BCACHE\_wbInvAll performs a global writeback and invalidate.

- ☐ All lines are invalidated in L1P cache.
- ☐ All lines are invalidated in the L1D cache and the data in the L1D cache is written back to L2 and/or external.
- ☐ All lines are invalidated in the L2 cache and the data in the L2 cache is written back to external.

This function always waits for other cache operations to finish before performing its actions. This function always waits until its writeback and invalidation operations are complete to return.

See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for more on cache writebacks.

**Constraints and  
Calling Context**

- ☐ none

**See Also**

BCACHE\_invL1pAll  
BCACHE\_wbAll  
BCACHE\_wbInv

## 2.3 BUF Module

The BUF module maintains buffer pools of fixed-size buffers.

### Functions

- ❑ BUF\_alloc. Allocate a fixed-size buffer from the buffer pool
- ❑ BUF\_create. Dynamically create a buffer pool
- ❑ BUF\_delete. Delete a dynamically-created buffer pool
- ❑ BUF\_free. Free a fixed-size buffer back to the buffer pool
- ❑ BUF\_maxbuff. Get the maximum number of buffers used in a pool
- ❑ BUF\_stat. Get statistics for the specified buffer pool

### Constants, Types, and Structures

```
typedef unsigned int MEM_sizep;

#define BUF_ALLOCSTAMP 0xcafe
#define BUF_FREESTAMP 0xbeef

typedef struct BUF_Obj {
    Ptr startaddr; /* Start addr of buffer pool */
    MEM_sizep size; /* Size before alignment */
    MEM_sizep postalignsize; /* Size after align */
    Ptr nextfree; /* Ptr to next free buffer */
    Uns totalbuffers; /* # of buffers in pool */
    Uns freebuffers; /* # of free buffers in pool */
    Int segid; /* Mem seg for buffer pool */
} BUF_Obj, *BUF_Handle;

typedef struct BUF_Attrs {
    Int segid; /* segment for element allocation */
} BUF_Attrs;

BUF_Attrs BUF_ATTRS = { /* default attributes */
    0,
};

typedef struct BUF_Stat {
    MEM_sizep postalignsize; /* Size after align */
    MEM_sizep size; /* Original size of buffer */
    Uns totalbuffers; /* Total buffers in pool */
    Uns freebuffers; /* # of free buffers in pool */
} BUF_Stat;
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the BUF Manager Properties and BUF Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

## Module Configuration Parameters

Name	Type	Default (Enum Options)
OBJMEMSEG	Reference	prog.get("IDRAM")

## Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
bufSeg	Reference	prog.get("IDRAM")
bufCount	Int32	1
size	Int32	8
align	Int32	4
len	Int32	8
postalignsize	Int32	8

## Description

The BUF module maintains pools of fixed-size buffers. These buffer pools can be created statically or dynamically. Dynamically-created buffer pools are allocated from a dynamic memory heap managed by the MEM module. Applications typically allocate buffer pools statically when size and alignment constraints are known at design time. Run-time allocation is used when these constraints vary during execution.

Within a buffer pool, all buffers have the same size and alignment. 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. You can create multiple buffer pools, each with a different buffer size.

Buffers can be allocated and freed from a pool as needed at run-time using the BUF\_alloc and BUF\_free functions.

The advantages of allocating memory from a buffer pool instead of from the dynamic memory heaps provided by the MEM module include:

- ❑ **Deterministic allocation times.** The BUF\_alloc and BUF\_free functions require a constant amount of time. Allocating and freeing memory through a heap is not deterministic.
- ❑ **Callable from all thread types.** Allocating and freeing buffers is atomic and non-blocking. As a result, BUF\_alloc and BUF\_free can be called from all types of DSP/BIOS threads: HWI, SWI, TSK, and IDL. In contrast, HWI and SWI threads cannot call MEM\_alloc.
- ❑ **Optimized for fixed-length allocation.** In contrast MEM\_alloc is optimized for variable-length allocation.

- ## BUF Manager Properties

- ❑ **Object Memory.** The memory segment to contain all BUF objects. (A BUF object may be stored in a different location than the buffer pool memory itself.)

Example: `bios.BUF.OBJMEMSEG = prog.get("myMEM");`

The following properties can be set for a buffer pool object in the BUF Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script. To create an BUF object in a configuration script, use the following syntax:

The Tconf examples that follow assume the object has been created as shown.

- Tconf Name: comment Type: String

- ❑ **Memory segment for buffer pool.** Select the memory segment in which the buffer pool is to be created. The linker decides where in the segment the buffer pool starts.

Tconf Name: bufSeg                      Type: Reference

Example: `myBuf.bufSeq = prog.get("myMEM");`

- Tconf Name: bufCount                      Type: Int32

Example: `myBuf.bufCount = 128;`

- Tconf Name: size Type: Int32

Example: `myBuf.size = 8;`

- ❑ **Buffer alignment.** Specify the alignment boundary for fixed-length buffers in the pool. Each buffer is aligned on boundaries with a multiple of this number. The default size shown is the minimum valid value for that platform. The value must be a power of 2.

Tconf Name: align

Type: Int32

Example: `myBuf.align = 4;`

- ❑ **Buffer pool length.** The actual length of the buffer pool (in MADUs) is calculated by multiplying the Buffer count by the Buffer size after alignment. You cannot modify this value directly.

Tconf Name: len

Type: Int32

Example: `myBuf.len = 8;`

- ❑ **Buffer size after alignment.** This property shows the modified Buffer size after applying the alignment. For example, if the Buffer size is 9 and the alignment is 4, the Buffer size after alignment is 12 (the next whole number multiple of 4 after 9).

Tconf Name: postalignsize

Type: Int32

Example: `myBuf.postalignsize = 8;`



**BUF\_alloc***Allocate a fixed-size buffer from a buffer pool***C Interface**

**Syntax**                      bufaddr = BUF\_alloc(buf);

**Parameters**                BUF\_Handle buf;        /\* buffer pool object handle \*/

**Return Value**              Ptr                bufaddr; /\* pointer to free buffer \*/

**Reentrant**                      yes

**Description**                BUF\_alloc allocates a fixed-size buffer from the specified buffer pool and returns a pointer to the buffer. BUF\_alloc does not initialize the allocated buffer space.

The buf parameter is a handle to identify the buffer pool object, from which the fixed size buffer is to be allocated. If the buffer pool was created dynamically, the handle is the one returned by the call to BUF\_create. If the buffer pool was created statically, the handle can be referenced as shown in the example that follows.

If buffers are available in the specified buffer pool, BUF\_alloc returns a pointer to the buffer. If no buffers are available, BUF\_alloc returns NULL.

The BUF module manages synchronization so that multiple threads can share the same buffer pool for allocation and free operations.

The time required to successfully execute BUF\_alloc is deterministic (constant over multiple calls).

**Example**

```
extern BUF_Obj bufferPool;
BUF_Handle buffPoolHandle = &bufferPool;

Ptr buffPtr;

/* allocate a buffer */
buffPtr = BUF_alloc(buffPoolHandle);
if (buffPtr == NULL) {
    SYS_abort("BUF_alloc failed");
}
```

**See Also**                      BUF\_free  
MEM\_alloc

**BUF\_create***Dynamically create a buffer pool***C Interface****Syntax**

```
buf = BUF_create(numbuff, size, align, attrs);
```

**Parameters**

```
Uns      numbuff; /* number of buffers in the pool */
MEM_sizep size;   /* size of a single buffer in the pool */
Uns      align;   /* alignment for each buffer in the pool */
BUF_Attrs *attrs; /* pointer to buffer pool attributes */
```

**Return Value**

```
BUF_Handle buf; /* buffer pool object handle */
```

**Reentrant**

```
no
```

**Description**

BUF\_create creates a buffer pool object dynamically. The parameters correspond to the properties available for statically-created buffer pools, which are described in the BUF Object Properties topic.

The numbuff parameter specifies how many fixed-length buffers the pool should contain. This must be a non-zero number.

The size parameter specifies how long each fixed-length buffer in the pool should be in MADUs. This must be a non-zero number. The size you specify is adjusted as needed to meet the alignment requirements, so the actual buffer size may be larger. The MEM\_sizep type is defined as follows:

```
typedef unsigned int MEM_sizep;
```

The align parameter specifies the alignment boundary for buffers in the pool. Each buffer is aligned on a boundary with an address that is a multiple of this number. The value must be a power of 2. The size of buffers created in the pool is automatically increased to accommodate the alignment you specify.

BUF\_create ensures that the size and alignment are set to at least the minimum values permitted for the platform. The minimum size permitted is 8 MADUs. The minimum alignment permitted is 4.

The attrs parameter points to a structure of type BUF\_Attrs, which is defined as follows:

```
typedef struct BUF_Attrs {
    Int    segid; /* segment for element allocation*/
} BUF_Attrs;
```

The segid element can be used to specify the memory segment in which buffer pool should be created. If attrs is NULL, the new buffer pool is created the default attributes specified in BUF\_ATTRS, which uses the default memory segment.

BUF\_create calls MEM\_alloc to dynamically create the BUF object's data structure and the buffer pool.

BUF\_create returns a handle to the buffer pool of type BUF\_Handle. If the buffer pool cannot be created, BUF\_create returns NULL. The pool may not be created if the numbuff or size parameter is zero or if the memory available in the specified heap is insufficient.

The time required to successfully execute BUF\_create is not deterministic (that is, the time varies over multiple calls).

### Constraints and Calling Context

- ❑ BUF\_create cannot be called from a SWI or HWI.
- ❑ The product of the size (after adjusting for the alignment) and numbuff parameters should not exceed the maximum Uns value.
- ❑ The alignment should be greater than the minimum value and must be a power of 2. If it is not, proper creation of buffer pool is not guaranteed.

### Example

```
BUF_Handle myBufpool;
BUF_Attrs myAttrs;

myAttrs = BUF_ATTRS;
myBufpool=BUF_create(5, 4, 2, &myAttrs);
if( myBufpool == NULL ){
    LOG_printf(&trace,"BUF_create failed!");
}
```

### See Also

BUF\_delete

**BUF\_delete***Delete a dynamically-created buffer pool***C Interface**

**Syntax**                    `status = BUF_delete(buf);`

**Parameters**              `BUF_Handle buf;        /* buffer pool object handle */`

**Return Value**            `Uns            status;    /* returned status */`

**Reentrant**                `no`

**Description**            BUF\_delete frees the buffer pool object and the buffer pool memory referenced by the handle provided.

The buf parameter is the handle that identifies the buffer pool object. This handle is the one returned by the call to BUF\_create. BUF\_delete cannot be used to delete statically created buffer pool objects.

BUF\_delete returns 1 if it has successfully freed the memory for the buffer object and buffer pool. It returns 0 (zero) if it was unable to delete the buffer pool.

BUF\_delete calls MEM\_free to delete the BUF object and to free the buffer pool memory. MEM\_free must acquire a lock to the memory before proceeding. If another task already holds a lock on the memory, there is a context switch.

The time required to successfully execute BUF\_delete is not deterministic (that is, the time varies over multiple calls).

**Constraints and  
Calling Context**

- ❑ BUF\_delete cannot be called from a SWI or HWI.
- ❑ BUF\_delete cannot be used to delete statically created buffer pool objects. No check is performed to ensure that this is the case.
- ❑ BUF\_delete assumes that all the buffers allocated from the buffer pool have been freed back to the pool.

**Example**

```
BUF_Handle myBufpool;
Uns delstat;

delstat = BUF_delete(myBufpool);
if( delstat == 0 ){
    LOG_printf(&trace, "BUF_delete failed!");
}
```

**See Also**                BUF\_create

**BUF\_free***Free a fixed memory buffer into the buffer pool***C Interface****Syntax**

```
status = BUF_free(buf, bufaddr);
```

**Parameters**

```
BUF_Handle buf;      /* buffer pool object handle */
Ptr         bufaddr; /* address of buffer to free */
```

**Return Value**

```
Bool         status; /* returned status */
```

**Reentrant**

```
yes
```

**Description**

BUF\_free frees the specified buffer back to the specified buffer pool. The newly freed buffer is then available for further allocation by BUF\_alloc.

The buf parameter is the handle that identifies the buffer pool object. This handle is the one returned by the call to BUF\_create.

The bufaddr parameter is the pointer returned by the corresponding call to BUF\_alloc.

BUF\_free always returns TRUE if DSP/BIOS real-time analysis is disabled (in the GBL Module Properties). If real-time analysis is enabled, BUF\_free returns TRUE if the bufaddr parameter is within the range of the specified buffer pool; otherwise it returns FALSE.

The BUF module manages synchronization so that multiple threads can share the same buffer pool for allocation and free operations.

The time required to successfully execute BUF\_free is deterministic (constant over multiple calls).

**Example**

```
extern BUF_Obj bufferPool;
BUF_Handle buffPoolHandle = &bufferPool;
Ptr buffPtr;
```

```
...
```

```
BUF_free(buffPoolHandle, buffPtr);
```

**See Also**

```
BUF_alloc
MEM_free
```

## BUF\_maxbuff

*Check the maximum number of buffers from the buffer pool*

### C Interface

<b>Syntax</b>	<code>count = BUF_maxbuff(buf);</code>
<b>Parameters</b>	<code>BUF_Handle buf;</code> <i>/* buffer pool object Handle */</i>
<b>Return Value</b>	<code>Uns            count;</code> <i>/*maximum number of buffers used */</i>

**Reentrant**                      no

**Description**                      BUF\_maxbuff returns the maximum number of buffers that have been allocated from the specified buffer pool at any time. The count measures the number of buffers in use, not the total number of times buffers have been allocated.

The buf parameter is the handle that identifies the buffer pool object. This handle is the one returned by the call to BUF\_create.

BUF\_maxbuff distinguishes free and allocated buffers via a stamp mechanism. Allocated buffers are marked with the BUF\_ALLOCSTAMP stamp (0xcafe). If the application happens to change this stamp to the BUF\_FREESTAMP stamp (0xbeef), the count may be inaccurate. Note that this is not an application error. This stamp is only used for BUF\_maxbuff, and changing it does not affect program execution.

The time required to successfully execute BUF\_maxbuff is not deterministic (that is, the time varies over multiple calls).

### Constraints and Calling Context

- ❑ BUF\_maxbuff cannot be called from a SWI or HWI.
- ❑ The application must implement synchronization to ensure that other threads do not perform BUF\_alloc during the execution of BUF\_maxbuff. Otherwise, the count returned by BUF\_maxbuff may be inaccurate.

### Example

```
extern BUF_Obj bufferPool;
BUF_Handle buffPoolHandle = &bufferPool;
Int maxbuff;

maxbuff = BUF_maxbuff(buffPoolHandle);
LOG_printf(&trace, "Max buffers used: %d", maxbuff);
```

### See Also

**BUF\_stat***Determine the status of a buffer pool***C Interface**

**Syntax** `BUF_stat(buf,statbuf);`

**Parameters** `BUF_Handle buf;` /\* buffer pool object handle \*/  
`BUF_Stat *statbuf;` /\* pointer to buffer status structure \*/

**Return Value** none

**Reentrant** yes

**Description**

BUF\_stat returns the status of the specified buffer pool.

The buf parameter is the handle that identifies the buffer pool object. This handle is the one returned by the call to BUF\_create.

The statbuf parameter must be a structure of type BUF\_Stat. The BUF\_stat function fills in all the fields of the structure. The BUF\_Stat type has the following fields:

```
typedef struct BUF_Stat {
    MEM_sizep postalignsize; /* Size after align */
    MEM_sizep size; /* Original size of buffer */
    Uns totalbuffers; /* Total # of buffers in pool */
    Uns freebuffers; /* # of free buffers in pool */
} BUF_Stat;
```

Size values are expressed in Minimum Addressable Data Units (MADUs). BUF\_stat collects statistics with interrupts disabled to ensure the correctness of the statistics gathered.

The time required to successfully execute BUF\_stat is deterministic (constant over multiple calls).

**Example**

```
extern BUF_Obj bufferPool;
BUF_Handle buffPoolHandle = &bufferPool;
BUF_Stat stat;

BUF_stat(buffPoolHandle, &stat);
LOG_printf(&trace, "Free buffers Available: %d",
           stat.freebuffers);
```

**See Also**

MEM\_stat

## 2.4 C62 and C64 Modules

The C62 and C64 modules include target-specific functions for the TMS320C6000 family. Use the C62 APIs for 'C62x, 'C67x, and 'C67+ devices. Use the 'C64 APIs for 'C64x and 'C64x+ devices.

### Functions

- ❑ C62\_disableIER. ASM macro to disable selected interrupts in IER
- ❑ C62\_enableIER. ASM macro to enable selected interrupts in IER
- ❑ C62\_plug. Plug interrupt vector
- ❑ C64\_disableIER. ASM macro to disable selected interrupts in IER
- ❑ C64\_enableIER. ASM macro to enable selected interrupts in IER
- ❑ C64\_plug. Plug interrupt vector

### Description

The C62 and C64 modules provide certain target-specific functions and definitions for the TMS320C6000 family of processors.

See the c62.h or c64.h files for a complete list of definitions for hardware flags for C. The c62.h and c64.h files contain C language macros, #defines for various TMS320C6000 registers, and structure definitions. The c62.h62 and c64.h64 files also contain assembly language macros for saving and restoring registers in HWIs.



**C62\_disableIER***Disable certain maskable interrupts***C Interface**

<b>Syntax</b>	<code>oldmask = C62_disableIER(mask);</code>
<b>Parameters</b>	Uns <code>mask;</code> <i>/* disable mask */</i>
<b>Return Value</b>	Uns <code>oldmask;</code> <i>/* actual bits cleared by disable mask */</i>

**Description**

C62\_disableIER disables interrupts by clearing the bits specified by mask in the Interrupt Enable Register (IER).

C62\_disableIER returns a mask of bits actually cleared. This return value should be passed to C62\_enableIER to re-enable interrupts.

See C62\_enableIER for a description and code examples for safely protecting a critical section of code from interrupts.

**See Also**

C62\_enableIER

**C64\_disableIER***Disable certain maskable interrupts***C Interface**

<b>Syntax</b>	<code>oldmask = C64_disableIER(mask);</code>
<b>Parameters</b>	Uns <code>mask;</code> <i>/* disable mask */</i>
<b>Return Value</b>	Uns <code>oldmask;</code> <i>/* actual bits cleared by disable mask */</i>

**Description**

C64\_disableIER disables interrupts by clearing the bits specified by mask in the Interrupt Enable Register (IER).

C64\_disableIER returns a mask of bits actually cleared. This return value should be passed to C64\_enableIER to re-enable interrupts.

See C64\_enableIER for a description and code examples for safely protecting a critical section of code from interrupts.

**See Also**

C64\_enableIER

**C62\_enableIER***Enable certain maskable interrupts***C Interface**

<b>Syntax</b>	C62_enableIER(oldmask);
<b>Parameters</b>	Uns            oldmask; /* enable mask */
<b>Return Value</b>	Void

**Description**

C62\_disableIER and C62\_enableIER disable and enable specific internal interrupts by modifying the Interrupt Enable Register (IER). C62\_disableIER clears the bits specified by the mask parameter in the IER and returns a mask of the bits it cleared. C62\_enableIER sets the bits specified by the oldmask parameter in the IER.

C62\_disableIER and C62\_enableIER are usually used in tandem to protect a critical section of code from interrupts. The following code examples show a region protected from all interrupts:

```
/* C example */
Uns    oldmask;

oldmask = C62_disableIER(~0);
    `do some critical operation; `
    `do not call TSK_sleep, SEM_post, etc.`
C62_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 C62\_disableIER / C62\_enableIER 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 / TSK\_enable. You can also use C62\_disableIER / C62\_enableIER 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 = C62_disableIER(INTMASK);
    `do some critical operation;`
    `NOT OK to call TSK_sleep, SEM_post, etc.`
C62_enableIER(oldmask);
TSK_enable();
```

---

**Note:**

If you use C\_disableIER / C62\_enableIER 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**

C62\_disableIER

**C64\_enableIER***Enable certain maskable interrupts***C Interface**

<b>Syntax</b>	C64_enableIER(oldmask);
<b>Parameters</b>	Uns            oldmask; /* enable mask */
<b>Return Value</b>	Void

**Description**

C64\_disableIER and C64\_enableIER are used to disable and enable specific internal interrupts by modifying the Interrupt Enable Register (IER). C64\_disableIER clears the bits specified by the mask parameter in the Interrupt Mask Register and returns a mask of the bits it cleared. C64\_enableIER sets the bits specified by the oldmask parameter in the Interrupt Mask Register.

C64\_disableIER and C64\_enableIER 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:

```
/* C example */
Uns    oldmask;

oldmask = C64_disableIER(~0);
    `do some critical operation; `
    `do not call TSK_sleep, SEM_post, etc.`
C64_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 C64\_disableIER and C64\_enableIER 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 / TSK\_enable. You can also use C64\_disableIER and C64\_enableIER 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 = C64_disableIER(INTMASK);
    `do some critical operation;`
    `NOT OK to call TSK_sleep, SEM_post, etc.`
C64_enableIER(oldmask);
TSK_enable();
```

---

**Note:**

If you use C64\_disableIER and C64\_enableIER 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**

C64\_disableIER

**C62\_plug***C function to plug an interrupt vector***C Interface****Syntax**

C62\_plug(vecid, fxn, dmachan);

**Parameters**

Int            vecid;     /\* interrupt id \*/  
 Fxn           fxn;       /\* pointer to HWI function \*/  
 Int           dmachan; /\* DMA channel to use for performing plug \*/

**Return Value**

Void

**Description**

C62\_plug writes an Interrupt Service Fetch Packet (ISFP) into the Interrupt Service Table (IST), at the address corresponding to vecid. The op-codes written in the ISFP create a branch to the function entry point specified by fxn:

```
stw    b0, *SP--[1]
mvk     fxn, b0
mvkh    fxn, b0
b       b0
ldw     *++SP[1], b0
nop     4
```

The dmachan necessary depends upon whether the IST is stored in internal or external RAM:

- ❑ **IST is in internal RAM.** If the CPU cannot access internal program RAM, a DMA channel must be used and the dmachan parameter must be a valid DMA channel. For example, 'C6x0x devices cannot access internal program RAM.

If the CPU can access internal program RAM, the dmachan parameter should be set to -1, which causes a CPU copy. For example, 'C6x11 devices can access internal program RAM.

- ❑ **IST is in external RAM.** The dmachan parameter should be set to -1.

If a DMA channel is specified by the dmachan parameter, C62\_plug assumes that the DMA channel is available for use, and stops the DMA channel before programming it. If the DMA channel is shared with other code, a semaphore or other DSP/BIOS signaling method should be used to provide mutual exclusion before calling C62\_plug.

C62\_plug does not enable the interrupt. Use C62\_enableIER to enable specific interrupts.

**Constraints and Calling Context**

- ❑ vecid must be a valid interrupt ID in the range of 0-15.
- ❑ dmachan must be 0, 1, 2, or 3 if the IST is in internal program memory and the device is a 'C6x0x.

**See Also**

C62\_enableIER  
 HWI\_dispatchPlug

**C64\_plug***C function to plug an interrupt vector***C Interface****Syntax**

C64\_plug(vecid, fxn, dmachan);

**Parameters**

Int            vecid;        /\* interrupt id \*/  
 Fxn           fxn;        /\* pointer to HWI function \*/  
 Int           dmachan; /\* DMA channel to use for performing plug \*/

**Return Value**

Void

**Description**

C64\_plug writes an Interrupt Service Fetch Packet (ISFP) into the Interrupt Service Table (IST), at the address corresponding to vecid. The op-codes written in the ISFP create a branch to the function entry point specified by fxn:

```
stw    b0, *SP-- [1]
mvk    fxn, b0
mvkh   fxn, b0
b      b0
ldw    *++SP [1], b0
nop    4
```

C64\_plug hooks up the specified function as the branch target for a hardware interrupt (fielded by the CPU) at the vector address corresponding to vecid. C64\_plug does not enable the interrupt. Use or C64\_enableIER to enable specific interrupts.

For C64x devices, you may set dmachan to -1 to specify a CPU copy, regardless of where the IST is stored. Alternately, you may specify a DMA channel. If you use dmachan to specify a DMA channel, C64\_plug assumes that the DMA channel is available for use, and stops the DMA channel before programming it. If the DMA channel is shared with other code, use a semaphore or other DSP/BIOS signaling method to provide mutual exclusion before calling C64\_plug or HWI\_dispatchPlug.

For C64x+ devices, dmachan is ignored. However, there is a case where DMA is automatically used on C64x+ devices. If the vector table location is L1P SRAM, then IDMA1 is used for the vector copy. In this case, the API waits for any activity to finish on IDMA1 before using it. It then waits for the vector copy DMA activity to complete before returning. Since the stack is used for the source location of the DMA copy, C64\_plug must be called while a stack from internal memory (L1 or L2) is active (and only when the vector table is in L1P SRAM).

**Constraints and Calling Context**

- ❑ vecid must be a valid interrupt ID in the range of 0-15.

**See Also**

C64\_enableIER  
 HWI\_dispatchPlug



## 2.5 CLK Module

The CLK module is the clock manager.

### Functions

- ❑ CLK\_countspms. Timer counts per millisecond
- ❑ CLK\_cpuCyclesPerHtime. Return high-res time to CPU cycles factor
- ❑ CLK\_cpuCyclesPerLtime. Return low-res time to CPU cycles factor
- ❑ CLK\_gethtime. Get high-resolution time
- ❑ CLK\_getltime. Get low-resolution time
- ❑ CLK\_getprd. Get period register value
- ❑ CLK\_reconfig. Reset timer period and registers using CPU frequency
- ❑ CLK\_start. Restart low-resolution timer
- ❑ CLK\_stop. Stop low-resolution timer

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the CLK Manager Properties and CLK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")
TIMERSELECT	String	"Timer 0"
ENABLECLK	Bool	true
HIRESTIME	Bool	true
ENABLEHTIME	Bool	true ('C64x+ only)
SPECIFYRATE	Bool	usually false ('C64x+ only)
INPUTCLK	Numeric	166.6667 (varies by platform)
RESETTIMER	Bool	true ('C64x+ only)
TIMMODE	EnumString	"32-bit unchained" ("32-bit chained", "externally programmed") ('C64x+ only)
MICROSECONDS	Int16	1000
CONFIGURETIMER	Bool	false
PRD	Int16	33250, 37500, or 75000 (varies by platform)

Name	Type	Default
TCRTDDR	EnumInt	0 (0 to 0xffffffff) ('C64x+ only)
POSTINITFXN	Extern	prog.extern("FXN_F_nop") ('C672x only)
CONONDEBUG	Bool	false ('C672x only)
STARTBOTH	Bool	false ('C672x only)

### Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
fxn	Extern	prog.extern("FXN_F_nop")
order	Int16	0

## Description

The CLK module provides methods for gathering timing information and for invoking functions periodically. The CLK module provides real-time clocks with functions to access the low-resolution and high-resolution times. These times can be used to measure the passage of time in conjunction with STS accumulator objects, as well as to add timestamp messages in event logs.

DSP/BIOS provides the following timing methods:

- ❑ **Timer Counter.** This DSP/BIOS counter changes at a relatively fast platform-specific rate. This counter is used only if the Clock Manager is enabled in the CLK Manager Properties.
- ❑ **Low-Resolution Time.** This time is incremented when the timer counter reaches its target value. When this time is incremented, any functions defined for CLK objects are run.
- ❑ **High-Resolution Time.** For some platforms, the timer counter is also used to determine the high-resolution time. For other platforms, a different timer is used for the high-resolution time.
- ❑ **Periodic Rate.** The PRD functions can be run at a multiple of the clock interrupt rate (the low-resolution rate) if you enable the "Use CLK Manager to Drive PRD" in the PRD Manager Properties.
- ❑ **System Clock.** The PRD rate, in turn, can be used to run the system clock, which is used to measure TSK-related timeouts and ticks. If you set the "TSK Tick Driven By" in the TSK Manager Properties to "PRD", the system clock ticks at the specified multiple of the clock interrupt rate (the low-resolution rate).

**Timer Counter**

The timer counter changes at a relatively fast rate until it reaches a target value. When the target value is reached, the timer counter is reset, a timer interrupt occurs, the low-resolution time is incremented, and any functions defined for CLK objects are run.

Table 2-1 shows the rate at which the timer counter changes, its target value, and how the value is reset once the target value has been reached.

*Table 2-1. Timer Counter Rates, Targets, and Resets*

Platform	Timer Counter Rate	Target Value	Value Reset
'C6201, 'C6211, 'C6713	Incremented every 4 CPU cycles.	PRD value	Counter reset to 0.
'C672x and devices with Real-Time Interrupt Module (RTI)	Incremented at $\text{SYSCLK} / 4$ .	Compare register value (same as PRD)	Counter reset to 0.
'C6416	Incremented every 8 CPU cycles.	PRD value	Counter reset to 0.
'C64x+	This rate is device-specific. On some devices, this rate is the CPU frequency divided by 6. On others, a dedicated 24 MHz or 27 MHz crystal drives this counter. Check the CLK manager statements in your configuration script for the frequency of this counter.	PRD value	Counter reset to 0.

**Low-Resolution Time**

When the value of the timer counter is reset to the value in the right-column of Table 2-1, the following actions happen:

- ❑ A timer interrupt occurs
- ❑ As a result of the timer interrupt, the HWI object for the selected timer runs the CLK\_F\_isr function.
- ❑ The CLK\_F\_isr function causes the low-resolution time to be incremented by 1.
- ❑ The CLK\_F\_isr function causes all the CLK Functions to be performed in sequence in the context of that HWI.

**Note: Specifying On-device Timer**

The configuration allows you to specify which on-device timer you want to use. DSP/BIOS requires the default setting in the Interrupt Selector Register for the selected timer. For example, interrupt 14 must be configured for Timer 0, interrupt 15 must be configured for Timer 1, and interrupt 11 must be configured for Timer 2.

Therefore, the low-resolution clock ticks at the timer interrupt rate and returns 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.

You can use `GBL_setFrequency`, `CLK_stop`, `CLK_reconfig`, and `CLK_start` to change the low-resolution timer rate.

The low-resolution time is stored as a 32-bit value. Its value restarts at 0 when the maximum value is reached.

On C64x+ devices, the low-resolution timer uses the least-significant 32 bits of the 64-bit GP/WD Timer. This time is configured for dual 32-bit unchained timer mode operation (except for some early C64x+ simulators). The upper 32 bits of the timer are available to your application. However, you must observe the following precautions:

- ❑ Do not modify TCR12, which controls the operation of the low-resolution timer.
- ❑ Do not modify PRD1 and PRD2, which determine the period of the low-resolution timer interrupt.
- ❑ Do not modify the TIMMODE nor the TIM12RS\_ bits of the Timer Global Control Register (TGCR).

**High-Resolution Time**

The high-resolution time is determined as follows for your platform:

*Table 2-2. High-Resolution Time Determination*

Platform	Description
'C6201, 'C6211, 'C6713	Number of times the timer counter has been incremented.
'C672x and devices with RTI module	Number of times the timer counter has been incremented.
'C6416	Number of times the timer counter has been incremented.

Table 2-2. High-Resolution Time Determination

Platform	Description
'C64x+	The TSCL register is used for this counter. The TSCL increments at the CPU frequency. For example, on a 1 GHz device, the TSCL increments at 1 GHz.

You can use the CLK\_gethptime 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 high-resolution time is stored as a 32-bit value. For platforms that use the same timer counter as the low-resolution time, the 32-bit high-resolution time is actually calculated by multiplying the low-resolution time by the value of the PRD property and adding number of timer counter increments since the last timer counter reset.

The high-resolution value restarts at 0 when the maximum value is reached.

CLK Functions

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 the HWI dispatcher when it runs CLK\_F\_isr. Additionally, CLK functions should **not** use the *interrupt* keyword or the INTERRUPT pragma in C functions.

The HWI object that runs the CLK\_F\_isr function is configured to use the HWI dispatcher. You can modify the dispatcher-specific properties of this HWI object. For example, you can change the interrupt mask value and the cache control value. See the HWI Module, page 2–173, for a description of the HWI dispatcher and these HWI properties. *You may not* disable the use of the HWI dispatcher for the HWI object that runs the CLK\_F\_isr function.

CLK Manager Properties

The following global properties can be set for the CLK module in the CLK Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment that contains the CLK objects created in the configuration.

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.CLK.OBJMEMSEG = prog.get("myMEM");`

- ❑ **CPU Interrupt.** Shows which HWI interrupt is used to drive the timer services. The value is changed automatically when you change the Timer Selection. This is an informational property only.

Tconf Name: N/A

- ❑ **Timer Selection.** The on-device timer to use. Changing this setting also automatically changes the CPU Interrupt used to drive the timer services and the function property of the relevant HWI objects.

Tconf Name: TIMERSELECT

Type: String

Options: "Timer 0", "Timer 1"

Example: `bios.CLK.TIMERSELECT = "Timer 0";`

- ❑ **Enable CLK Manager.** If this property is set to true, the on-device timer hardware is used to drive the high- and low-resolution times and to trigger execution of CLK functions. On platforms where the separate ENABLEHTIME property is available, setting the ENABLECLK property to true and the ENABLEHTIME property to false enables only the low-resolution timer.

Tconf Name: ENABLECLK

Type: Bool

Example: `bios.CLK.ENABLECLK = true;`

- ❑ **Use high resolution time for internal timings.** If this property is set to true, the high-resolution timer is used to monitor internal periods. Otherwise the less intrusive, low-resolution timer is used.

Tconf Name: HIRESTIME

Type: Bool

Example: `bios.CLK.HIRESTIME = true;`

- ❑ **Enable high resolution timer.** If this property is set to true, this parameter enables the high-resolution timer. This property is available only for the 'C64x+. For platforms that use only one timer, the high-resolution and low-resolution timers are both enabled and disabled by the "Enable CLK Manager" property.

Tconf Name: ENABLEHTIME

Type: Bool

Example: `bios.CLK.ENABLEHTIME = true;`

- ❑ **Specify input clock rate.** (C64x+ only) If this property is set to true, you can specify the Input frequency (INPUTCLK) property. Otherwise, the default clock frequency is used. The default is dependant on the platform.

Tconf Name: SPECIFYRATE

Type: Bool

Example: `bios.CLK.SPECIFYRATE = true;`

- ❑ **Input frequency (MHz).** Set this property to the low-resolution timer's input clock frequency. This is the timer used by CLK\_gettime.

Tconf Name: INPUTCLK

Type: Numeric

Example: `bios.CLK.INPUTCLK = 166.6667;`

- ❑ **Reset Timer and TIMMODE.** (C64x+ only) This property is checked by the DSP/BIOS CLK startup code. If it is set to true (the default), DSP/BIOS initializes the timer to the mode specified by the Timer Mode (TIMMODE) property. This property is provided because some applications share the timer with DSP/BIOS and need to assume responsibility for initializing the timer mode. On some DaVinci devices, for example, the ARM operating system and DSP/BIOS share a timer. In this configuration, the ARM is responsible for initializing and resetting the timer, so you should set the RESETTIMER property for DSP/BIOS to false.

Tconf Name: RESETTIMER

Type: Bool

Example: `bios.CLK.RESETTIMER = true;`

- ❑ **Timer Mode.** (C64x+ only) This property specifies the timer mode. There is no benefit to selecting either the "32-bit chained" or "externally programmed" mode. These modes are provided only for some early C64x+ simulators that did not support 32-bit unchained mode. See "Low-Resolution Time" on page 2-53 for more about the timer mode.

Tconf Name: TIMMODE

Type: EnumString

Options: "32-bit chained", "32-bit unchained", and "externally programmed"

Example: `bios.CLK.TIMMODE = "32-bit unchained";`

- ❑ **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.

Tconf Name: MICROSECONDS

Type: Int16

Example: `bios.CLK.MICROSECONDS = 1000;`

- ❑ **Directly configure on-device timer registers.** If this property is set to true, the period register can be directly set to the desired value. In this case, the Microseconds/Int property is computed based on the value in period register and the CPU clock speed in the GBL Module Properties.

Tconf Name: CONFIGURETIMER Type: Bool

Example: `bios.CLK.CONFIGURETIMER = false;`

- ❑ **TDDR register.** The value of the on-device timer prescaler.

Platform	Options	Size	Registers
'C64x+	00h to 0ffffffh	32 bits	PRD3:PRD4

Tconf Name: TCRTDDR Type: EnumInt

Example: `bios.CLK.TCRTDDR = 2;`

- ❑ **PRD Register.** This value specifies the interrupt period and is used to configure the PRD register. The default value varies depending on the platform.

Tconf Name: PRD Type: Int16

Example: `bios.CLK.PRD = 33250;`

- ❑ **Timer 1 Init Function.** ('C672x and RTI timer module devices only)  
This function runs during the DSP/BIOS timer startup process. It is intended to be used to perform Timer 1 setup. This function should set all Timer 1 related registers and should enable the Timer 1 interrupt in the IER. The sequence of events performed during the CLK module startup is as follows:

- Perform Timer 0 setup.
- Set the COMP1 and CPUC1 registers to the same value as the COMP0 and CPUC0 registers.
- Call the Timer 1 Init Function specified by this property.
- Enable the Timer 0 interrupt and start Timer 0. If the "Start Both Timer 0 and Timer 1" property is true, Timer 1 is also enabled and started.

Tconf Name: POSTINITFXN Type: Extern

Example: `bios.CLK.POSTINITFXN =  
          prog.extern("FXN_F_nop");`



- ❑ **Continue Counting in Debug Mode.** ('C672x and RTI timer module devices only) If this property is set to true, the timer counter continues to count in debug mode even when the program is halted at a breakpoint.

Tconf Name: CONONDEBUG

Type: Bool

Example: `bios.CLK.CONONDEBUG = false;`

- ❑ **Start Both Timer 0 and Timer 1.** ('C672x and RTI timer module devices only) If this property is set to true, DSP/BIOS starts both Timer 0 and timer 1 during the DSP/BIOS CLK module startup. This causes the Timer 0 clock and the Timer 1 clock to be synchronized.

Tconf Name: STARTBOTH

Type: Bool

Example: `bios.CLK.STARTBOTH = false;`

- ❑ **Instructions/Int.** The number of instruction cycles represented by the period specified above. This is an informational property only.

Tconf Name: N/A

## CLK Object Properties

The Clock Manager allows you to create an arbitrary number of CLK objects. Clock objects have functions, which are 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.

To create a CLK object in a configuration script, use the following syntax:

```
var myClk = bios.CLK.create("myClk");
```

The following properties can be set for a clock function object in the CLK Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. The Tconf examples assume the myClk object has been created as shown.

- ❑ **comment.** Type a comment to identify this CLK object.

Tconf Name: comment

Type: String

Example: `myClk.comment = "Runs timeFxn";`

- ❑ **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 and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. (The DSP/BIOS Configuration Tool generates assembly code, which must use leading underscores when referencing C functions or labels.) If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

Tconf Name: `fxn` Type: Extern

Example: `myClk.fxn = prog.extern("timeFxn");`

- ❑ **order.** You can change the sequence in which CLK functions are executed by specifying the order property of all the CLK functions.

Tconf Name: `order` Type: Int16

Example: `myClk.order = 2;`

**CLK\_countspms***Number of hardware timer counts per millisecond***C Interface****Syntax**                      ncounts = CLK\_countspms();**Parameters**                  Void**Return Value**                LgUns              ncounts;**Reentrant**                    yes**Description**                CLK\_countspms returns the number of high-resolution timer counts per millisecond.

CLK\_countspms can be used to compute an absolute length of time from the number of low resolution timer interrupts. For example, the following code computes time in milliseconds.

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

The equation below computes time in milliseconds since the last wrap of the high-resolution timer counter.

```
timeAbs = CLK_gethtime() / CLK_countspms();
```

**See Also**

CLK\_gethtime  
 CLK\_getprd  
 CLK\_cpuCyclesPerHtime  
 CLK\_cpuCyclesPerLtime  
 GBL\_getClkin  
 STS\_delta

**CLK\_cpuCyclesPerHtime**    *Return multiplier for converting high-res time to CPU cycles*

**C Interface**

**Syntax**                      `ncycles = CLK_cpuCyclesPerHtime(Void);`

**Parameters**                `Void`

**Return Value**              `Float`              `ncycles;`

**Reentrant**                   `yes`

**Description**                CLK\_cpuCyclesPerHtime returns the multiplier required to convert from high-resolution time to CPU cycles. High-resolution time is returned by CLK\_gettime.

For example, the following code returns the number of CPU cycles and the absolute time elapsed during processing.

```
time1 = CLK_gettime();
... processing ...
time2 = CLK_gettime();
CPUCycles = (time2 - time1) * CLK_cpuCyclesPerHtime();
/* calculate absolute time in milliseconds */
TimeAbsolute = CPUCycles / GBL_getFrequency();
```

**See Also**                    CLK\_gettime  
                              CLK\_getprd  
                              GBL\_getClkin

**CLK\_cpuCyclesPerLtime**    *Return multiplier for converting low-res time to CPU cycles*

## C Interface

**Syntax**                      `ncycles = CLK_cpuCyclesPerLtime(Void);`

**Parameters**                `Void`

**Return Value**              `Float`                `ncycles;`

**Reentrant**                    `yes`

**Description**                CLK\_cpuCyclesPerLtime returns the multiplier required to convert from low-resolution time to CPU cycles. Low-resolution time is returned by CLK\_gettime.

For example, the following code returns the number of CPU cycles and milliseconds elapsed during processing.

```
time1 = CLK_gettime();
... processing ...
time2 = CLK_gettime();
CPUCycles = (time2 - time1) * CLK_cpuCyclesPerLtime();
/* calculate absolute time in milliseconds */
TimeAbsolute = CPUCycles / GBL_getFrequency();
```

**See Also**                    CLK\_gettime  
                                 CLK\_getprd  
                                 GBL\_getClkin

**CLK\_gethtime***Get high-resolution time***C Interface****Syntax** `currtime = CLK_gethtime();`**Parameters** `Void`**Return Value** `LgUns currtime /* high-resolution time */`**Reentrant** `no`**Description**

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

See “High-Resolution Time” on page 2-54 for information about how this rate is set.

CLK\_gethtime provides a value with greater accuracy than CLK\_gettime, but which wraps back to 0 more frequently. For example, if the timer tick 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.

**Constraints and Calling Context**

- ❑ CLK\_gethtime cannot be called from the program’s main() function.

**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**

**Syntax** `currtime = CLK_gettime();`

**Parameters** `Void`

**Return Value** `LgUns currtime /* low-resolution time */`

**Reentrant** `yes`

**Description**

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. See “Low-Resolution Time” on page 2-53 for information about the other 32 bits of this value for C64x+ devices.

The low-resolution time is the number of timer interrupts that have occurred. See “Low-Resolution Time” on page 2-53 for information about how this rate is set.

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.

CLK\_gethtime provides a value with more accuracy than CLK\_gettime, but which wraps back to 0 more frequently. For example, if the timer tick rate is 200 MHz, and you use the default period register value of 50000, the CLK\_gethtime value wraps back to 0 approximately every 86 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.

**Constraints and Calling Context**

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

**Example**

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

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

**See Also**

CLK\_gethtime  
PRD\_getticks  
STS\_delta

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

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

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

<b>Description</b>	CLK_getprd returns the number of high-resolution timer counts per low-resolution interrupt.
--------------------	---

CLK\_getprd can be used to compute an absolute length of time from the number of low-resolution timer interrupts. For example, the following code computes time in milliseconds.

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

**See Also**

CLK\_countspms  
CLK\_gettime  
CLK\_cpuCyclesPerHtime  
CLK\_cpuCyclesPerLtime  
GBL\_getClkin  
STS\_delta



**CLK\_reconfig***Reset timer period and registers using current CPU frequency***C Interface**

<b>Syntax</b>	status = CLK_reconfig();
<b>Parameters</b>	Void
<b>Return Value</b>	Bool            status     /* FALSE if failed */
<b>Reentrant</b>	yes

**Description**

This function needs to be called after a call to GBL\_setFrequency. It computes values for the timer period and the prescalar registers using the new CPU frequency. The new values for the period and prescalar registers ensure that the CLK interrupt runs at the statically configured interval in microseconds.

The return value is FALSE if the timer registers cannot accommodate the current frequency or if some other internal error occurs.

When calling CLK\_reconfig outside of main(), you must also call CLK\_stop and CLK\_start to stop and restart the timer. Use the following call sequence:

```
/* disable interrupts if an interrupt could lead to
   another call to CLK_reconfig or if interrupt
   processing relies on having a running timer */
HWI_disable() or SWI_disable()
GBL_setFrequency(cpuFreqInKhz);
CLK_stop();
CLK_reconfig();
CLK_start();
HWI_restore() or SWI_enable()
```

When calling CLK\_reconfig from main(), the timer has not yet been started. (The timer is started as part of BIOS\_startup(), which is called internally after main.) As a result, you can use the following simplified call sequence in main():

```
GBL_setFrequency(cpuFreqInKhz);
CLK_reconfig(Void);
```

Note that GBL\_setFrequency does not affect the PLL, and therefore has no effect on the actual frequency at which the DSP is running. It is used only to make DSP/BIOS aware of the DSP frequency you are using.

## Constraints and Calling Context

- ❑ When calling CLK\_reconfig from anywhere other than main(), you must also use CLK\_stop and CLK\_start.
- ❑ Call HWI\_disable/HWI\_restore or SWI\_disable/SWI\_enable around a block that stops, configures, and restarts the timer as needed to prevent re-entrancy or other problems. That is, you must disable interrupts if an interrupt could lead to another call to CLK\_reconfig or if interrupt processing relies on having a running timer to ensure that these non-reentrant functions are not interrupted.
- ❑ If you do not stop and restart the timer, CLK\_reconfig can only be called from the program's main() function.
- ❑ If you use CLK\_reconfig, you should also use GBL\_setFrequency.

## See Also

GBL\_getFrequency  
GBL\_setFrequency  
CLK\_start  
CLK\_stop

**CLK\_start***Restart the low-resolution timer***C Interface****Syntax** CLK\_start();**Parameters** Void**Return Value** Void**Reentrant** no**Description**

This function starts the low-resolution timer if it has been halted by CLK\_stop. The period and prescaler registers are updated to reflect any changes made by a call to CLK\_reconfig. This function then resets the timer counters and starts the timer.

CLK\_start should only be used in conjunction with CLK\_reconfig and CLK\_stop. See the section on CLK\_reconfig for details and the allowed calling sequence.

Note that all 'C6000 platforms except the 'C64x+ use the same timer to drive low-resolution and high-resolution times. On such platforms, both times are affected by this API.

- ❑ Call HWI\_disable/HWI\_restore or SWI\_disable/SWI\_enable around a block that stops, configures, and restarts the timer as needed to prevent re-entrancy or other problems. That is, you must disable interrupts if an interrupt could lead to another call to CLK\_start or if interrupt processing relies on having a running timer to ensure that these non-reentrant functions are not interrupted
- ❑ This function cannot be called from main().

**See Also**

CLK\_reconfig  
CLK\_stop  
GBL\_setFrequency

**CLK\_stop***Halt the low-resolution timer***C Interface****Syntax** CLK\_stop();**Parameters** Void**Return Value** Void**Reentrant** no**Description**

This function stops the low-resolution timer. It can be used in conjunction with CLK\_reconfig and CLK\_start to reconfigure the timer at run-time.

Note that all 'C6000 platforms except the 'C64x+ use the same timer to drive low-resolution and high-resolution times. On such platforms, both times are affected by this API.

CLK\_stop should only be used in conjunction with CLK\_reconfig and CLK\_start, and only in the required calling sequence. See the section on CLK\_reconfig for details.

- ❑ Call HWI\_disable/HWI\_restore or SWI\_disable/SWI\_enable around a block that stops, configures, and restarts the timer as needed to prevent re-entrancy or other problems. That is, you must disable interrupts if an interrupt could lead to another call to CLK\_stop or if interrupt processing relies on having a running timer to ensure that these non-reentrant functions are not interrupted
- ❑ This function cannot be called from main().

**See Also**

CLK\_reconfig  
CLK\_start  
GBL\_setFrequency

## 2.6 DEV Module

The DEV module provides the device interface.

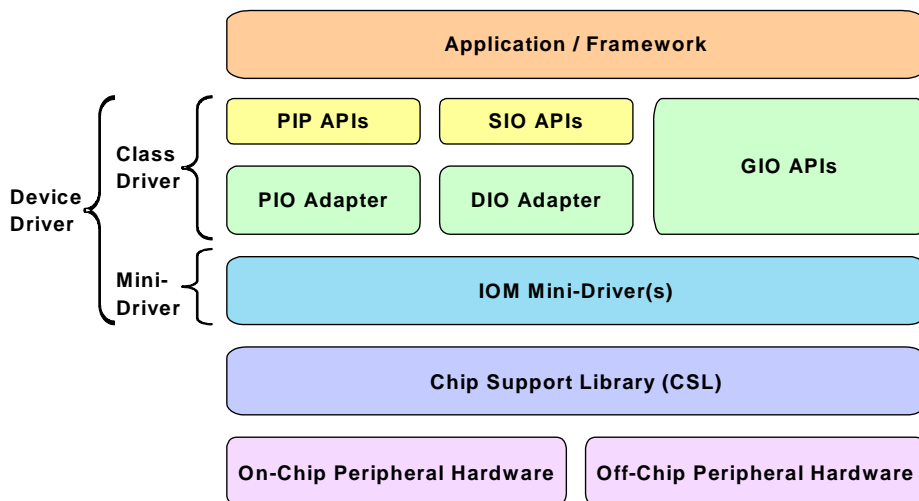
### Functions

- ❑ DEV\_createDevice. Dynamically create device
- ❑ DEV\_deleteDevice. Delete dynamically-created device
- ❑ 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

### Description

DSP/BIOS provides two device driver models that enable applications to communicate with DSP peripherals: IOM and SIO/DEV.

The components of the IOM model are illustrated in the following figure. It separates hardware-independent and hardware-dependent layers. Class drivers are hardware independent; they manage device instances, synchronization and serialization of I/O requests. The lower-level mini-driver is hardware-dependent. See the *DSP/BIOS Driver Developer's Guide* (SPRU616) for more information on the IOM model.



The SIO/DEV model provides a streaming I/O interface. In this model, the application indirectly invokes DEV functions implemented by the driver managing the physical device attached to the stream, using generic functions provided by the SIO module. See the *DSP/BIOS User's Guide* (SPRU423) for more information on the SIO/DEV model.

The model used by a device is identified by its function table type. A type of IOM\_Fxns is used with the IOM model. A type of DEV\_Fxns is used with the DEV/SIO model.

The DEV module provides the following capabilities:

- ❑ **Device object creation.** You can create device objects through static configuration or dynamically through the DEV\_createDevice function. The DEV\_deleteDevice and DEV\_match functions are also provided for managing device objects.
- ❑ **Driver function templates.** The Dxx functions listed as part of the DEV module are templates for driver functions. These are the functions you create for drivers that use the DEV/SIO model.

## 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 */
    size_t    size;        /* buffer size */
    Arg       misc;        /* reserved for driver */
    Arg       arg;         /* user argument */
    Uns       cmd;         /* mini-driver command */
    Int       status;      /* status of command */
} DEV_Frame;

typedef struct DEV_Obj { /* device object */
    QUE_Handle todevice; /* downstream frames here */
    QUE_Handle fromdevice; /* upstream frames here */
    size_t     bufsize; /* buffer size */
    Uns        nbufs; /* number of buffers */
    Int        segid; /* buffer segment ID */
    Int        mode; /* DEV_INPUT/DEV_OUTPUT */
    Int        devid; /* device ID */
    Ptr        params; /* device parameters */
    Ptr        object; /* ptr to dev instance obj */
    DEV_Fxns   fxns; /* driver functions */
    Uns        timeout; /* SIO_reclaim timeout value */
    Uns        align; /* buffer alignment */
    DEV_Callback *callback; /* pointer to callback */
} 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 );
    size_t   (*reclaim)( DEV_Handle );
} DEV_Fxns;

typedef struct DEV_Callback {
    Fxn      fxn;         /* function */
    Arg      arg0;        /* argument 0 */
    Arg      arg1;        /* argument 1 */
} DEV_Callback;

typedef struct DEV_Device { /* device specifier */
    String   name;        /* device name */
    Void *   fxns;        /* device function table*/
    Int      devid;       /* device ID */
    Ptr      params;      /* device parameters */
    Uns      type;        /* type of the device */
    Ptr      devp;        /* pointer to device handle */
} DEV_Device;

typedef struct DEV_Attrs {
    Int      devid;       /* device id */
    Ptr      params;      /* device parameters */
    Uns      type;        /* type of the device */
    Ptr      devp;        /* device global data ptr */
} DEV_Attrs;

```

## Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DEV Manager Properties and DEV Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

### Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
initFxn	Arg	0x00000000
fxnTable	Arg	0x00000000
fxnTableType	EnumString	"DEV_Fxns" ("IOM_Fxns")
deviceld	Arg	0x00000000
params	Arg	0x00000000
deviceGlobalDataPtr	Arg	0x00000000

## DEV Manager Properties

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

- ❑ **DGN Driver (software generator driver).** 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 Driver (host link driver).** Driver that uses the HST interface to send data to and from the Host Channel Control Analysis Tool.
- ❑ **DIO Adapter (class driver).** Driver used with the device driver model.
- ❑ **DPI Driver (pipe driver).** Software device used to stream data between DSP/BIOS tasks.

To configure devices for other drivers, use Tconf to create a User-defined Device (UDEV) object. There are no global properties for the user-defined device manager.

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

- ❑ **DGS Driver.** Stackable gather/scatter driver
- ❑ **DNL Driver.** Null driver
- ❑ **DOV Driver.** Stackable overlap driver
- ❑ **DST Driver.** Stackable "split" driver
- ❑ **DTR Driver.** Stackable streaming transformer driver

## DEV Object Properties

The following properties can be set for a user-defined device in the UDEV Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. To create a user-defined device object in a configuration script, use the following syntax:

```
var myDev = bios.UDEV.create("myDev");
```

The Tconf examples assume the myDev object is created as shown.

- ❑ **comment.** Type a comment to identify this object.  
Tconf Name: comment Type: String  
Example: `myDev.comment = "My device";`
- ❑ **init function.** Specify the function to run to initialize this device.  
Use a leading underscore before the function name if the function is written in C and you are using the DSP/BIOS Configuration Tool. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.  
Tconf Name: initFxn Type: Arg  
Example: `myDev.initFxn = prog.extern("myInit");`



- ❑ **function table ptr.** Specify the name of the device functions table for the driver or mini-driver. This table is of type DEV\_Fxns or IOM\_Fxns depending on the setting for the function table type property.

Tconf Name: fxnTable

Type: Arg

Example: `myDev.fxnTable =  
          prog.extern("mydevFxnTable");`

- ❑ **function table type.** Choose the type of function table used by the driver to which this device interfaces. Use the IOM\_Fxns option if you are using the DIO class driver to interface to a mini-driver with an IOM\_Fxns function table. Otherwise, use the DEV\_Fxns option for other drivers that use a DEV\_Fxns function table and Dxx functions. You can create a DIO object only if a UDEV object with the IOM\_Fxns function table type exists.

Tconf Name: fxnTableType

Type: EnumString

Options: "DEV\_Fxns", "IOM\_Fxns"

Example: `myDev.fxnTableType = "DEV_Fxns";`

- ❑ **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.

Tconf Name: deviceId

Type: Arg

Example: `myDev.deviceId = prog.extern("devID");`

- ❑ **device params ptr.** 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 if the structure is declared in C and you are using the DSP/BIOS Configuration Tool.

Tconf Name: params

Type: Arg

Example: `myDev.params = prog.extern("myParams");`

- ❑ **device global data ptr.** Provide a pointer to any global data to be used by this device. This value can be set only if the function table type is IOM\_Fxns.

Tconf Name: deviceGlobalDataPtr

Type: Arg

Example: `myDev.deviceGlobalDataPtr = 0x00000000;`

**DEV\_createDevice***Dynamically create device***C Interface****Syntax**

```
status = DEV_createDevice(name, fxns, initFxn, attrs);
```

**Parameters**

```
String      name;      /* name of device to be created */
Void        *fxns;     /* pointer to device function table */
Fxn         initFxn;   /* device init function */
DEV_Attrs   *attrs;    /* pointer to device attributes */
```

**Return Value**

```
Int         status;    /* result of operation */
```

**Reentrant**

```
no
```

**Description**

DEV\_createDevice allows an application to create a user-defined device object at run-time. The object created has parameters similar to those defined statically for the DEV Object Properties. After being created, the device can be used as with statically-created DEV objects.

The name parameter specifies the name of the device. The device name should begin with a slash (/) for consistency with statically-created devices and to permit stacking drivers. For example "/codec" might be the name. The name must be unique within the application. If the specified device name already exists, this function returns failure.

The fxns parameter points to the device function table. The function table may be of type DEV\_Fxns or IOM\_Fxns.

The initFxn parameter specifies a device initialization function. The function passed as this parameter is run if the device is created successfully. The initialization function is called with interrupts disabled. If several devices may use the same driver, the initialization function (or a function wrapper) should ensure that one-time initialization actions are performed only once.

The attrs parameter points to a structure of type DEV\_Attrs. This structure is used to pass additional device attributes to DEV\_createDevice. If attrs is NULL, the device is created with default attributes. DEV\_Attrs has the following structure:

```
typedef struct DEV_Attrs {
    Int      devid;    /* device id */
    Ptr      params;   /* device parameters */
    Uns      type;     /* type of the device */
    Ptr      devp;     /* device global data ptr */
} DEV_Attrs;
```

The `devid` item specifies 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. The default value is `NULL`.

The `params` item specifies the name of a parameter structure that may be used to provide additional parameters. 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. The default value is `NULL`.

The `type` item specifies the type of driver used with this device. The default value is `DEV_IOMTYPE`. The options are:

Type	Use With
<code>DEV_IOMTYPE</code>	Mini-drivers used in the IOM model.
<code>DEV_SIOTYPE</code>	DIO adapter with SIO streams or other DEV/SIO drivers

The `devp` item specifies the device global data pointer, which points to any global data to be used by this device. This value can be set only if the table type is `IOM_Fxns`. The default value is `NULL`.

If an `initFxn` is specified, that function is called as a result of calling `DEV_createDevice`. In addition, if the device type is `DEV_IOMTYPE`, the `mdBindDev` function in the function table pointed to by the `fxns` parameter is called as a result of calling `DEV_createDevice`. Both of these calls are made with interrupts disabled.

`DEV_createDevice` returns one of the following status values:

Constant	Description
<code>SYS_OK</code>	Success.
<code>SYS_EINVAL</code>	A device with the specified name already exists.
<code>SYS_EALLOC</code>	The heap is not large enough to allocate the device.

`DEV_createDevice` calls `SYS_error` if `mdBindDev` returns a failure condition. The device is not created if `mdBindDev` fails, and `DEV_createDevice` returns the IOM error returned by the `mdBindDev` failure.

## Constraints and Calling Context

- ❑ This function cannot be called from a SWI or HWI.
- ❑ This function can only be used if dynamic memory allocation is enabled.

- ❑ The device function table must be consistent with the type specified in the attrs structure. DSP/BIOS does not check to ensure that the types are consistent.

**Example**

```
Int status;

/* Device attributes of device "/pipe0" */
DEV_Attrs dpiAttrs = {
    NULL,
    NULL,
    DEV_SIOTYPE,
    0
};

status = DEV_createDevice("/pipe0", &DPI_FXNS,
    (Fxn)DPI_init, &dpiAttrs);
if (status != SYS_OK) {
    SYS_abort("Unable to create device");
}
```

**See Also**

SIO\_create

**DEV\_deleteDevice***Delete a dynamically-created device***C Interface**

**Syntax**                    `status = DEV_deleteDevice(name);`

**Parameters**             `String        name;        /* name of device to be deleted */`

**Return Value**           `Int            status;       /* result of operation */`

**Reentrant**                `no`

**Description**            DEV\_deleteDevice deallocates the specified dynamically-created device and deletes it from the list of devices in the application.

The name parameter specifies the device to delete. This name must match a name used with DEV\_createDevice.

Before deleting a device, delete any SIO streams that use the device. SIO\_delete cannot be called after the device is deleted.

If the device type is DEV\_IOMTYPE, the mdUnBindDev function in the function table pointed to by the fxns parameter of the device is called as a result of calling DEV\_deleteDevice. This call is made with interrupts disabled.

DEV\_createDevice returns one of the following status values:

Constant	Description
SYS_OK	Success.
SYS_ENODEV	No device with the specified name exists.

DEV\_deleteDevice calls SYS\_error if mdUnBindDev returns a failure condition. The device is deleted even if mdUnBindDev fails, but DEV\_deleteDevice returns the IOM error returned by mdUnBindDev.

**Constraints and Calling Context**

- ☐ This function cannot be called from a SWI or HWI.
- ☐ This function can be used only if dynamic memory allocation is enabled.
- ☐ The device name must match a dynamically-created device. DSP/BIOS does not check that the device was not created statically.

**Example**                    `status = DEV_deleteDevice("/pipe0");`

**See Also**                  SIO\_delete

**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 */
```

**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. The DEV\_Device structure is defined in dev.h.

The substr return value contains a pointer to the characters remaining after the match. 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*


---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**C Interface****Syntax**

```
status = Dxx_close(device);
```

**Parameters**

```
DEV_Handle device;    /* device handle */
```

**Return Value**

```
Int                status;    /* result of operation */
```

**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

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

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 */
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	<ul style="list-style-type: none"><li>❑ device must be bound to a device by a prior call to Dxx_open.</li></ul>
See Also	SIO_ctrl



**Dxx\_idle***Idle device*


---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 */
```

**Description**

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.

Dxx\_idle returns the device to its initial state. Dxx\_idle should move any frames from the device->toqueue queue to the device->fromqueue queue. In SIO\_ISSUERECLAIM mode, any outstanding buffers issued to the stream must be reclaimed in order to return the device to its true initial state.

Dxx\_idle is called by SIO\_idle, SIO\_flush, and SIO\_delete to recycle frames to the appropriate queue.

flush is a boolean parameter that indicates what to do with any pending data of an output stream. 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.

**Constraints and Calling Context**

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

**See Also**

```
SIO_delete
SIO_idle
SIO_flush
```

Dxx\_init

Initialize device

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

C Interface

Syntax	Dxx_init();
Parameters	Void
Return Value	Void
Description	<p>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.</p> <p>Dxx_init is called at system startup, before the application's main() function is called.</p>

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


---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**C Interface****Syntax**

```
status = Dxx_issue(device);
```

**Parameters**

```
DEV_Handle device;    /* device handle */
```

**Return Value**

```
Int          status;   /* result of operation */
```

**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, 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.

**Constraints and  
Calling Context**

**See Also**

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.

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

Dxx\_reclaim

SIO\_issue

**Dxx\_open***Open device*

---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**C Interface****Syntax**

```
status = Dxx_open(device, name);
```

**Parameters**

```
DEV_Handle device;    /* driver handle */  
String      name;     /* device name */
```

**Return Value**

```
Int         status;    /* result of operation */
```

**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*

---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**C Interface****Syntax**

```
status = Dxx_ready(device, sem);
```

**Parameters**

```
DEV_Handle device;    /* device handle */  
SEM_Handle sem;       /* semaphore to post when ready */
```

**Return Value**

```
Bool                status;    /* TRUE if device is ready */
```

**Description**

Dxx\_ready is called by SIO\_select and SIO\_ready 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.

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.

SIO\_ready calls Dxx\_ready with sem = NULL. This is equivalent to the second Dxx\_ready call made by SIO\_select, and the underlying device driver should just return status without registering a semaphore.

**See Also**

SIO\_select

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


---

**Important Note:** This API will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**C Interface****Syntax**

```
status = Dxx_reclaim(device);
```

**Parameters**

```
DEV_Handle device;    /* device handle */
```

**Return Value**

```
Int                status;    /* result of operation */
```

**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



**DGN Driver***Software generator driver*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**Description**

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.

**Configuring a DGN Device**

To create a DGN device object in a configuration script, use the following syntax:

```
var myDgn = bios.DGN.create("myDgn");
```

See the DGN Object Properties for the device you created.

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DGN Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Instance Configuration Parameters**

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
device	EnumString	"user" ("sine", "random", "constant", "printHex", "printInt", "printFloat" ('C67x only))
useDefaultParam	Bool	false
deviceId	Arg	prog.extern("DGN_USER", "asm")
constant	Numeric	1 (1.0 for 'C67x)
seedValue	Int32	1
lowerLimit	Numeric	-32767 (0.0 for 'C67x)
upperLimit	Numeric	32767 (1.0 for 'C67x)
gain	Numeric	32767 (1.0 for 'C67x)
frequency	Numeric	1 (1000.0 for 'C67x)
phase	Numeric	0 (0.0 for 'C67x)

Name	Type	Default (Enum Options)
rate	Int32	256 (44000 for 'C67x)
fxn	Extern	prog.extern("FXN_F_nop")
arg	Arg	0x00000000

## Data Streaming

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.

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.

## DGN Driver Properties

There are no global properties for the DGN driver manager.

## DGN Object Properties

The following properties can be set for a DGN device on the DGN Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. To create a DGN device object in a script, use the following syntax:

```
var myDgn = bios.DGN.create("myDgn");
```

The Tconf examples assume the myDgn object is created as shown.

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

Tconf Name: comment

Type: String

Example: myDgn.comment = "DGN device";

- ❑ **Device category.** The device category—user, sine, random, constant, printHex, printInt, and printFloat ('C67x only)—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 or printFloat device can be opened for output data streaming only.

- **user.** Uses custom function to produce/consume a data stream.

- **sine.** Produce a stream of sine wave samples.

- **random.** Produces a stream of random values.

- **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.

■ **printFloat.** Writes the stream data buffers to the trace buffer in float format. ('C67x only)

Tconf Name: device Type: EnumString

Options: "user", "sine", "random", "constant", "printHex",  
"printInt", "printFloat" ('C67x only)

Example: `myDgn.device = "user";`

- ❑ **Use default parameters.** Set this property to true if you want to use the default parameters for the Device category you selected.

Tconf Name: useDefaultParam Type: Bool

Example: `myDgn.useDefaultParam = false;`

- ❑ **Device ID.** This property is set automatically when you select a Device category.

Tconf Name: deviceId Type: Arg

Example: `myDgn.deviceId = prog.extern("DGN_USER",  
"asm");`

- ❑ **Constant value.** The constant value to be generated if the Device category is constant.

Tconf Name: constant Type: Numeric

Example: `myDgn.constant = 1;`

- ❑ **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.

Tconf Name: seedValue Type: Int32

Example: `myDgn.seedValue = 1;`

- ❑ **Lower limit.** The lowest value to be generated if the Device category is random.

Tconf Name: lowerLimit Type: Numeric

Example: `myDgn.lowerLimit = -32767;`

- ❑ **Upper limit.** The highest value to be generated if the Device category is random.

Tconf Name: upperLimit Type: Numeric

Example: `myDgn.upperLimit = 32767;`

- ❑ **Gain.** The amplitude scaling factor of the generated sine wave if the Device category is sine. This 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.

Tconf Name: gain

Type: Numeric

Example: `myDgn.gain = 32767;`

- ❑ **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 for stepping through this table:

`step = (256 * Frequency / Rate)`

Tconf Name: frequency

Type: Numeric

Example: `myDgn.frequency = 1;`

- ❑ **Phase.** The phase of the generated sine wave (in radians) if the Device category is sine.

Tconf Name: phase

Type: Numeric

Example: `myDgn.phase = 0;`

- ❑ **Sample rate.** The sampling rate of the generated sine wave (in sample points per second) if the Device category is sine.

Tconf Name: rate

Type: Int32

Example: `myDgn.rate = 256;`

- ❑ **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 and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

Tconf Name: fxn

Type: Extern

Example: `myDgn.fxn = prog.extern("usrFxn");`

- ❑ **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.

Tconf Name: arg

Type: Arg

Example: `myDgn.arg = prog.extern("myArg");`

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


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 create a DGS device object in a configuration script, use the following syntax:

```
var myDgs = bios.UDEV.create("myDgs");
```

Modify the myDgs properties as follows.

- ☐ **init function.** Type 0 (zero).
- ☐ **function table ptr.** Type \_DGS\_FXNS
- ☐ **function table type.** DEV\_Fxns
- ☐ **device id.** Type 0 (zero).
- ☐ **device params ptr.** 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.

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 Tconf. To do that, add two new SIO objects. Enter /codec (or any other configured terminal device) as the Device Control String 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 String. 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 Driver



**DHL Driver***Host link driver*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 create an HST object and make it available to the DHL driver. To do this, use the following syntax:

```
var myHst = bios.HST.create("myHst");
myHst.availableForDHL = true;
```

Also be sure to set the mode property to "output" or "input" as needed by the DHL device. For example:

```
myHst.mode = "output";
```

Once there are HST channels available for DHL, you can create a DHL device object in a configuration script using the following syntax:

```
var myDhl = bios.DHL.create("myDhl");
```

Then, you can set this object's properties 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.

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DHL Driver Properties and DHL Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
hstChannel	Reference	prog.get("myHST")
mode	EnumString	"output" ("input")

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 statically-created stream, set the deviceName property of the SIO object to match the name of the DHL device you configured.

```
mySio.deviceName = prog.get("myDhl");
```

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

```
stream = SIO_create("/dhl0", 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 DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **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 using HST Manager Properties and HST Object Properties.

Tconf Name: OBJMEMSEG

Type: Reference

Example: `DHL.OBJMEMSEG = prog.get("myMEM");`

## DHL Object Properties

The following properties can be set for a DHL device using the DHL Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. To create a DHL device object in a configuration script, use the following syntax:

```
var myDhl = bios.DHL.create("myDhl");
```

The Tconf examples assume the myDhl object has been created as shown.

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

Tconf Name: comment

Type: String

Example: `myDhl.comment = "DHL device";`

- ❑ **Underlying HST Channel.** Select the underlying HST channel from the drop-down list. The "Make this channel available for a new DHL device" property in the HST Object Properties must be set to true for that HST object to be known here.

Tconf Name: hstChannel

Type: Reference

Example: `myDhl.hstChannel = prog.get("myHST");`

- ❑ **Mode.** This informational property shows the mode (input or output) of the underlying HST channel. This becomes the mode of the DHL device.

Tconf Name: mode

Type: EnumString

Options: "input", "output"

Example: `myDhl.mode = "output";`

**DIO Adapter***SIO Mini-driver adapter***Description**

The DIO adapter allows GIO-compliant mini-drivers to be used through SIO module functions. Such mini-drivers are described in the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

**Configure Mini-driver**

To create a DIO device object in a configuration script, first use the following syntax:

```
var myUdev = bios.UDEV.create("myUdev");
```

Set the DEV Object Properties for the device as follows.

- ☐ **init function.** Type 0 (zero).
- ☐ **function table ptr.** Type \_DIO\_FXNS
- ☐ **function table type.** IOM\_Fxns
- ☐ **device id.** Type 0 (zero).
- ☐ **device params ptr.** Type 0 (zero).

Once there is a UDEV object with the IOM\_Fxns function table type in the configuration, you can create a DIO object with the following syntax and then set properties for the object:

```
var myDio = bios.Dio.create("myDio");
```

**DIO Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DIO Driver Properties and DIO Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")
STATICCREATE	Bool	false

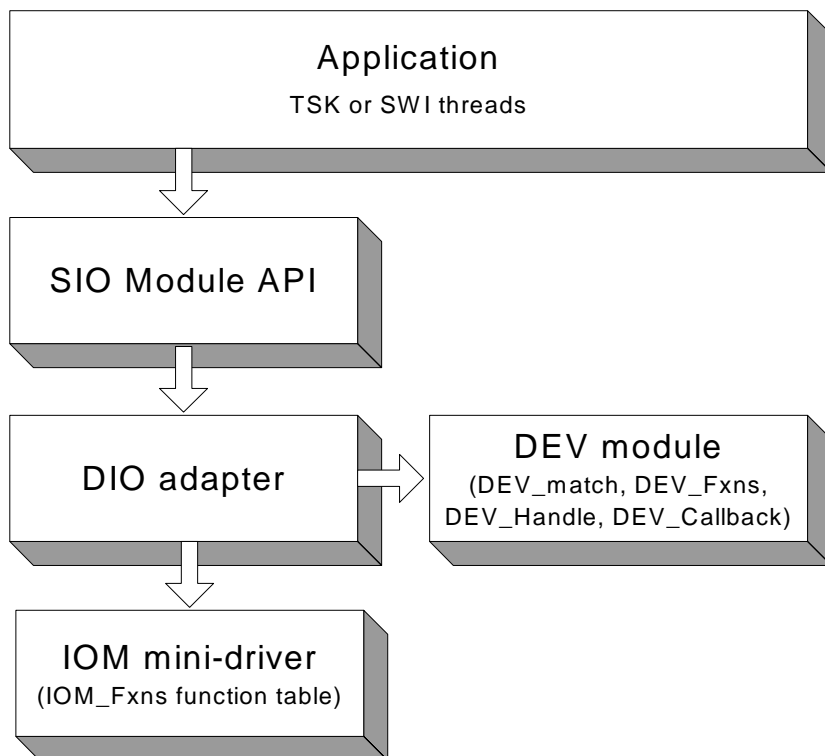
**Instance Configuration Parameters**

Name	Type	Default
comment	String	"<add comments here>"
useCallBackFxn	Bool	false
deviceName	Reference	prog.get("UDEVO")
chanParams	Arg	0x00000000

### Description

The mini-drivers described in the *DSP/BIOS Device Driver Developer's Guide* (SPRU616) are intended for use with the GIO module. However, the DIO driver allows them to be used with the SIO module instead of the GIO module.

The following figure summarizes how modules are related in an application that uses the DIO driver and a mini-driver:



## DIO Driver Properties

The following global properties can be set for the DIO - Class Driver on the DIO Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

- [illegible]

calls to SIO\_create fail. Setting this property to true reduces the application's code size (so long as the application does not call MEM\_alloc or its related functions elsewhere).

Tconf Name: STATICCREATE

Type: Bool

Example: `bios.DIO.STATICCREATE = false;`

## DIO Object Properties

The following properties can be set for a DIO device using the DIO Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. To create a DIO device object in a configuration script, use the following syntax:

```
var myDio = bios.DIO.create("myDio");
```

The Tconf examples assume the myDio object has been created as shown.

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

Tconf Name: comment

Type: String

Example: `myDio.comment = "DIO device";`

- ❑ **use callback version of DIO function table.** Set this property to true if you want to use DIO with a callback function. Typically, the callback function is SWI\_andnHook or a similar function that posts a SWI. Do not set this property to true if you want to use DIO with a TSK thread.

Tconf Name: useCallBackFxn

Type: Bool

Example: `myDio.useCallBackFxn = false;`

- ❑ **fxnsTable.** This informational property shows the DIO function table used as a result of the settings in the "use callback version of DIO function table" and "Create ALL DIO Objects Statically" properties. The four possible setting combinations of these two properties correspond to the four function tables: DIO\_tskDynamicFxns, DIO\_tskStaticFxns, DIO\_cbDynamicFxns, and DIO\_cbStaticFxns.

Tconf Name: N/A

- ❑ **device name.** Name of the device to use with this DIO object.

Tconf Name: deviceName

Type: Reference

Example: `myDio.deviceName = prog.get("UDEVO");`

- ❑ **channel parameters.** This property allows you to pass an optional argument to the mini-driver create function. See the chanParams parameter of the GIO\_create function.

Tconf Name: chanParams

Type: Arg

Example: `myDio.chanParams = 0x00000000;`

**DNL Driver***Null driver*

---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 create a DNL device object in a configuration script, use the following syntax:

```
var myDnl = bios.UDEV.create("myDnl");
```

Set DEV Object Properties for the device you created as follows.

- ❑ **init function.** Type 0 (zero).
- ❑ **function table ptr.** Type \_DNL\_FXNS
- ❑ **function table type.** DEV\_Fxns
- ❑ **device id.** Type 0 (zero).
- ❑ **device params ptr.** Type 0 (zero).

**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*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 8-bit word in the data address space of the processor on C6x platforms.

**Configuring a DOV Device**

To create a DOV device object in a configuration script, use the following syntax:

```
var myDov = bios.UDEV.create("myDov");
```

Set the DEV Object Properties for the device you created as follows.

- ☐ **init function.** Type 0 (zero).
- ☐ **function table ptr.** Type `_DOV_FXNS`
- ☐ **function table type.** `DEV_Fxns`
- ☐ **device id.** Type 0 (zero).
- ☐ **device params ptr.** Type 0 (zero) or the length of the overlap as described after this list.

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 statically create the stream (with `Tconf`) instead, enter the length of the overlap in the Device Control String for the stream.

For example, if you statically create a device called `overlap`, and use 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 the SIO Module 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 statically and enter 16 as the Device ID property, leave the Device Control String blank.

In addition to the configuration 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 statically (rather than dynamically with SIO\_create), add SIO objects with Tconf. Enter the string that identifies the terminating device preceded by "/" (forward slash) in the SIO object's Device Control Strings (for example, /codec, /port). Then select the stacking device (overlap, overlapio) from the Device property.

## See Also

DTR Driver  
DGS Driver

**DPI Driver***Pipe driver*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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.

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DPI Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Instance Configuration Parameters**

Name	Type	Default
comment	String	"<add comments here>"
allowVirtual	Bool	false

**Data Streaming**

After adding a DPI device called pipe0 in the configuration, 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 Tconf 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);
```

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.

## **DPI and the SIO\_ISSUERECLAIM Streaming Model**

In the SIO\_ISSUERECLAIM streaming model, an application reclaims 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 default one. To do this, add this version of dpi.obj to your project explicitly. 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 statically with Tconf 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 in the DPI Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script. To create a DPI device object in a configuration script, use the following syntax:

```
var myDpi = bios.DPI.create("myDpi");
```

The Tconf examples assume the myDpi object has been created as shown.

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

Tconf Name: comment

Type: String

Example: `myDpi.comment = "DPI device";`

- ☐ **Allow virtual instances of this device.** Set this property to true if you want to be able to use SIO\_create to dynamically create multiple streams to use this DPI device. DPI devices are used by SIO stream objects, which you create with Tconf or the SIO\_create function.

If this property is set to true, 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 property is set to `false`, 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` fails.

Tconf Name: `allowVirtual`

Type: Bool

Example: `myDpi.allowVirtual = false;`

**DST Driver***Stackable split driver*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 create a DST device object in a configuration script, use the following syntax:

```
var myDst = bios.UDEV.create("myDst");
```

Set the DEV Object Properties for the device you created as follows.

- ☐ **init function.** Type 0 (zero).
- ☐ **function table ptr.** Type `_DST_FXNS`
- ☐ **function table type.** `DEV_Fxns`
- ☐ **device id.** Type 0 (zero) or the number of small buffers corresponding to a large buffer as described after this list.
- ☐ **device params ptr.** Type 0 (zero).

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 `Tconf`, 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 Tconf (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 String. 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 Tconf, 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 String for an SIO object created with Tconf. Type only/codec for the Device Control String.

**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*


---

**Important Note:** This driver will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the IOM driver interface instead. See the *DSP/BIOS Driver Developer's Guide* (SPRU616).

---

**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 create a DTR device object in a configuration script, use the following syntax:

```
var myDtr = bios.UDEV.create("myDtr");
```

Set the DEV Object Properties for the device you created as follows.

- ❑ **init function.** Type 0 (zero).
- ❑ **function table ptr.** Type `_DTR_FXNS`
- ❑ **function table type.** `DEV_Fxns`
- ❑ **device id.** Type 0 (zero), `_DTR_multiply`, or `_DTR_multiplyInt16`.

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 built-in data scaling operation is performed on the data stream to multiply the contents of the buffer by the `scale.value` of the device parameters.

If you type `_DTR_multiplyInt16`, a built-in data scaling operation is performed on the data stream to multiply the contents of the buffer by the `scale.value` of the device parameters. The data stream is assumed to contain values of type `Int16`. This API is provided for fixed-point processors only.

- ❑ **device params ptr.** Enter the name of a DTR\_Params structure declared in your C application code. See the information following this list for details.

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.

If you do not configure one of the built-in scaling functions for the device ID, use `user.fxn` and `user.arg` in the DTR\_Params structure to 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 the SIO Module. A task can, of course, block as required by the underlying device.

## 2.7 ECM Module

The ECM module is the Event Combiner Manager for C64x+ devices, which have maskable (customizable) CPU interrupts.

### Functions

- ❑ ECM\_disableEvent. Disable the specified event at run-time.
- ❑ ECM\_dispatch. Handle events from the event combiner.
- ❑ ECM\_dispatchPlug. Create an ECM dispatcher table entry.
- ❑ ECM\_enableEvent. Enable the specified event at run-time.

### Constants, Types, and Structures

```
typedef struct ECM_Attrs {
    Arg arg;      /* function argument */
    Bool unmask; /* unmask == 1 means enable event */
} ECM_Attrs;
```

```
typedef Void (*ECM_Fxn) (Arg);
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the ECM Manager Properties section. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default
ENABLE	Bool	false

#### Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
fxn	Extern	prog.extern("_UTL_halt")
arg	Arg	0x00000000
unmask	Bool	false

### Description

The ECM module provides an interface to the C64x+ interrupt controller. This controller supports up to 128 system events. There are 12 maskable CPU interrupts (and their corresponding pins). The "interrupt selector" allows you to route any of the 128 system events to any maskable CPU interrupt. In addition, an "event combiner" allows you to combine up to 32 system events into a single event that is routed to a single CPU interrupt.

DSP/BIOS supports the C64x+ interrupt selector through the HWI Module. You can route one of the 128 system events to a specific HWI object by specifying the event number as the "interrupt selection number" in the HWI Object Properties. This one-to-one mapping supports up to 12 maskable interrupts.

See the "System Event Mapping" table in the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about interrupt selection numbers and their corresponding events. In addition, the device-specific data manual contains information about events listed as "Available events" in the table in SPRU871.

If the 16 HWI objects are sufficient for the number of HWI functions your application needs to run, you need not enable the ECM module. You do not need to know whether your C function will be run by the HWI module or ECM module when you write the function. (The ECM module uses the HWI dispatcher, and so its functions cannot be completely written in assembly.)

DSP/BIOS additionally supports the C64x+ event combiner by adding the ECM module to extend HWI functionality. In the DSP/BIOS Configuration Tool, the ECM manager is nested within the HWI manager. The ECM module allows you to specify the function and argument to be used when one of these system events is triggered.

The first four ECM events (0-3) are used to tie ECM events to HWI objects. The HWI objects that have an interrupt selection number from 0 to 3 run flagged (pending) events in the corresponding event combiner group if the ECM manager is enabled.

To combine events, do the following:

- 1) Set "Enable event combiner manager" in the ECM Manager Properties to true.
- 2) Set "unmask event source" in the ECM Object Properties to true for events you want to run in a combined event. The events are described in the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871).
- 3) Specify the function and any argument for each ECM event you unmask. By default, all ECM events run UTL\_halt (which runs an infinite loop with all processor interrupts disabled) and pass their event number as an argument.
- 4) Write your ECM functions just as you would if an HWI object were running the function. The HWI dispatcher is used internally to run ECM functions, so your function should be written in C/C++.

- 5) In the HWI Object Properties for a particular object, assign the interrupt selection number for the group of unmasked ECM events you want it to run. For example, if you assign an interrupt selection number of 3 to HWI\_INT10, that interrupt runs all unmasked ECM events that have been received (flagged) in the range of 96 to 127. HWI objects that run a combined event call the ECM\_dispatch function.

Table 2–3 ECM Events

Interrupt Selection Number	ECM Module Objects	ECM Event Range
0	EVENT4 to EVENT31	4 - 31
1	EVENT32 to EVENT63	32 - 63
2	EVENT64 to EVENT95	64 - 95
3	EVENT96 to EVENT127	96 - 127

- 6) Set "Use Dispatcher" to true for the HWI object that runs the combined event.

You can use the APIs in the ECM module to enable and disable ECM events at run-time and to handle combined events.

See the ECM\_dispatch topic description for more about how ECM groups are triggerred and run.

ECM Manager Properties

The following global properties can be set for the ECM module in the ECM Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ☐ **Enable event combiner manager.** Set this property to true to enable use of the ECM module.  
Tconf Name: ENABLE  
Type: Bool  
Example: `bios.ECM.ENABLE = false;`

ECM Object Properties

The following properties can be set for an ECM object in the ECM Object Properties dialog in the DSP/BIOS Configuration Tool or a Tconf script. You cannot create or delete ECM objects.

- ☐ **comment.** A comment to identify this ECM object.  
Tconf Name: comment  
Type: String  
Example: `bios.ECM.instance("EVENT4").comment = "event for combiner";`

- ❑ **function.** The function to execute for this system event. This function must be written in C (or be a C function that calls assembly), but must not call the HWI\_enter/HWI\_exit macro pair. Write this function as if it were an HWI function that used the HWI dispatcher. This function can post a SWI, but the SWI will not run until all the combined events have finished running.

Tconf Name: fxn

Type: Extern

Example: `bios.ECM.instance("EVENT4").fxn =  
          prog.extern("myEvent4");`

- ❑ **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.

Tconf Name: arg

Type: Arg

Example: `bios.ECM.instance("EVENT4").arg = 3;`

- ❑ **unmask event source.** Set this property to true to enable this event within its corresponding combined event (HWI interrupt selection numbers 0 to 3).

Tconf Name: unmask

Type: Bool

Example: `bios.ECM.instance("EVENT4").unmask =  
          true;`

**ECM\_disableEvent***Disable a system event in its event combiner mask***C Interface**

<b>Syntax</b>	ECM_disableEvent(eventID);
<b>Parameters</b>	Uns            eventID; /* individual event number from 4 to 127 */
<b>Return Value</b>	Void

**Description**

This function is available only for C64x+ devices, which have an event combiner for CPU interrupts.

This function sets the Event Mask bit that corresponds to the specified eventID to disabled (0). If you use this function, when the combined event that contains this individual event is run, the function for this individual event will not run, even though the event has occurred.

Information about the function and argument assigned to the event is still stored. You can use this function to temporarily disable individual events at run-time.

**Constraints and  
Calling Context**

❑ none

**Example**

```
ECM_disableEvent(42);
```

**See Also**

ECM\_enableEvent  
ECM\_dispatchPlug



**ECM\_dispatch***Run functions for a combined event***C Interface**

<b>Syntax</b>	ECM_dispatch(eventID);
<b>Parameters</b>	Uns            eventID; /* event number from 0 to 3 */
<b>Return Value</b>	Void

**Description**

This function is available only for C64x+ devices, which have an event combiner for CPU interrupts.

This function runs a combined event. That is, it runs all enabled and flagged system events within the range that corresponds to the eventID specified. If the ECM manager is enabled, this is the default function used by any HWI objects that have an interrupt selection number from 0 to 3.

Here is an example of the steps that occur when an ECM group is processed:

- 1) Suppose event 14 (an IDMA channel 1 interrupt) occurs. This flags EVENT14 in the Event Flag register so that it is marked pending.
- 2) Suppose that the ECM module is enabled and EVENT14 is unmasked (enabled).
- 3) EVENT14 is in event combiner group 0, which combines EVENT4 through EVENT31.
- 4) The occurrence of EVENT14 causes an interrupt for its associated HWI object. Any unmasked event in the combiner group would also trigger that HWI object.
- 5) The HWI object runs ECM\_dispatch with an argument of 0.
- 6) ECM\_dispatch makes a copy of the list of unmasked and flagged events in event combiner group 0. There may be more events than EVENT14 that are pending by the time this check is made. (The numeric order of the CPU interrupts from low to high determines the priority for processing HWI interrupts.)
- 7) The ECM manager runs functions for events that are: in the combiner group range, are unmasked, and have been flagged as pending.

The events that meet this criteria have their functions run from left to right (high to low numbers) in the register. There is no way to set priorities amongst combined events. The set of functions run to completion without preemption.

- 8) Since other interrupts for combined events can occur while the ECM manager is running a combined event, the ECM manager next checks to see if any events in the same combiner group have occurred during processing. If so, it processes those events by repeating the previous step.

The ECM\_dispatch function uses the HWI dispatcher internally.

### **Constraints and Calling Context**

- ❑ ECM\_dispatch should be called only as the function for an HWI object that has an interrupt selection number of 0 through 3. As such, it is always called in the context of an HWI.

### **Example**

```
ECM_dispatch(2);
```

### **See Also**

ECM\_dispatchPlug

**ECM\_dispatchPlug***Specify function and attributes for a system event***C Interface****Syntax**

ECM\_dispatchPlug(eventID, fxn, \*attrs);

**Parameters**

Uns            eventID;   /\* event number from 4 to 127 \*/  
 ECM\_Fxn      fxn;       /\* function to be plugged \*/  
 ECM\_Attrs    \*attrs     /\* attributes \*/

**Return Value**

Void

**Description**

This function is available only for C64x+ devices, which have an event combiner for CPU interrupts.

This function places an entry in a table used by the ECM manager that specifies the function, arg, and unmask properties for a particular ECM object. This allows run-time changes to the values in the static configuration of an ECM object.

The types used in the parameters to this function are defined as follows.

```
typedef Void (*ECM_Fxn) (Arg);

typedef struct ECM_Attrs {
    Arg arg;       /* function argument */
    Bool unmask; /* unmask == 1 means enable event */
} ECM_Attrs;
```

The specified system event is enabled by this function only if you set unmask to 1 in the ECM\_Attrs structure and if you have performed the other steps to enable the ECM manager and map an eventID from 0 to 3 to an HWI object's interrupt selection number.

**Constraints and Calling Context**

□ none

**Example**

```
ECM_Attrs ecmattrs = ECM_ATTRS;
ecmattrs.unmask = 1;

ECM_dispatchPlug(4, (Fxn)isrfunc, &ecmattrs);
C64_enable(0x10);
```

**See Also**

ECM\_disableEvent  
 ECM\_enableEvent

**ECM\_enableEvent***Enable a system event in its event combiner mask***C Interface**

<b>Syntax</b>	ECM_enableEvent(eventID);
<b>Parameters</b>	Uns            eventID; /* event number from 4 to 127 */
<b>Return Value</b>	Void

**Description**

This function is available only for C64x+ devices, which have an event combiner for CPU interrupts.

This function sets the Event Mask bit that corresponds to the specified eventID to enabled (1). If you previously used ECM\_disableEvent for this event, information about the function and argument assigned to the event is still retained. You can use this function to temporarily enable individual events at run-time.

The function for this event does not actually run until this individual event has occurred and its event combiner group is triggered.

**Constraints and  
Calling Context**

❑ none

**Example**

```
ECM_enableEvent(42);
```

**See Also**

ECM\_disableEvent  
ECM\_dispatchPlug

## 2.8 GBL Module

This module is the global settings manager.

### Functions

- ❑ GBL\_getClkin. Gets configured value of board input clock in KHz.
- ❑ GBL\_getFrequency. Gets current frequency of the CPU in KHz.
- ❑ GBL\_getProcid. Gets configured processor ID used by MSGQ.
- ❑ GBL\_getVersion. Gets DSP/BIOS version information.
- ❑ GBL\_setFrequency. Set frequency of CPU in KHz for DSP/BIOS.
- ❑ GBL\_setProcid. Set configured value of processor ID.

### Configuration Properties

The following list shows the properties for this module that can be configured in a Tconf script, along with their types and default values. For details, see the GBL Module Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default (Enum Options)
BOARDNAME	String	"c6xxx"
PROCID	Int16	0
CLKIN	UInt32	20000 KHz
CLKOUT	Int16	'C6201: 133.00 'C6211: 150 'C64x: 600 'C67x: 300 'C64x+: 1 'C672x: 300
SPECIFYRTSLIB	Bool	false
RTSLIB	String	""
ENDIANMODE	EnumString	"little" ("big")
CALLUSERINITFXN	Bool	false
USERINITFXN	Extern	prog.extern("FXN_F_nop")
ENABLEINST	Bool	true
INSTRUMENTED	Bool	true
ENABLEALLTRC	Bool	true
CSRPCC	EnumString	"mapped" ("cache enable", "cache freeze", "cache bypass")
C621XCONFIGUREL2	Bool	false

Name	Type	Default (Enum Options)
C641XCONFIGUREL2	Bool	false
C621XCCFGL2MODE	EnumString	"SRAM" ("1-way cache", "2-way cache", "3-way cache", "4-way cache")
C641XCCFGL2MODE	EnumString	"4-way cache (0k)" ("4-way cache (32k)", "4-way cache (64k)", "4-way cache (128k)", "4-way cache (256k)")
C621XMAR	Numeric	0x0000
C641XMAREMIFB	Numeric	0x0000
C641XMARCE0	Numeric	0x0000
C641XMARCE1	Numeric	0x0000
C641XMARCE2	Numeric	0x0000
C641XMARCE3	Numeric	0x0000
C641XCCFGP	EnumString	"urgent" ("high", "medium", "low")
C641XSETL2ALLOC	Bool	false
C641XL2ALLOC0	EnumInt	6
C641XL2ALLOC1	EnumInt	2 (0 to 7)
C641XL2ALLOC2	EnumInt	2 (0 to 7)
C641XL2ALLOC3	EnumInt	2 (0 to 7)
C64PLUSCONFIGURE	Bool	false
C64PLUSL1PCFG	EnumString	32k ("0k", "4k", "8k", "16k", "32k")
C64PLUSL1DCFG	EnumString	32k ("0k", "4k", "8k", "16k", "32k")
C64PLUSL2CFG	EnumString	0k ("0k", "32k", "64k", "128k", "256k")
C64PLUSMAR0to31	Numeric	0x0
C64PLUSMAR32to63	Numeric	0x0
C64PLUSMAR64to95	Numeric	0x0
C64PLUSMAR96to127	Numeric	0x0
C64PLUSMAR128to159	Numeric	0x0
C64PLUSMAR160to191	Numeric	0x0
C64PLUSMAR192to223	Numeric	0x0
C64PLUSMAR224to255	Numeric	0x0

Name	Type	Default (Enum Options)
C641XCONFIGUREL2	Bool	false
C621XCCFGL2MODE	EnumString	"SRAM" ("1-way cache", "2-way cache", "3-way cache", "4-way cache")
C641XCCFGL2MODE	EnumString	"4-way cache (0k)" ("4-way cache (32k)", "4-way cache (64k)", "4-way cache (128k)", "4-way cache (256k)")
C621XMAR	Numeric	0x0000
C641XMAREMIFB	Numeric	0x0000
C641XMARCE0	Numeric	0x0000
C641XMARCE1	Numeric	0x0000
C641XMARCE2	Numeric	0x0000
C641XMARCE3	Numeric	0x0000
C641XCCFGP	EnumString	"urgent" ("high", "medium", "low")
C641XSETL2ALLOC	Bool	false
C641XL2ALLOC0	EnumInt	6
C641XL2ALLOC1	EnumInt	2 (0 to 7)
C641XL2ALLOC2	EnumInt	2 (0 to 7)
C641XL2ALLOC3	EnumInt	2 (0 to 7)
C64PLUSCONFIGURE	Bool	false
C64PLUSL1PCFG	EnumString	32k ("0k", "4k", "8k", "16k", "32k")
C64PLUSL1DCFG	EnumString	32k ("0k", "4k", "8k", "16k", "32k")
C64PLUSL2CFG	EnumString	0k ("0k", "32k", "64k", "128k", "256k")
C64PLUSMAR0to31	Numeric	0x0
C64PLUSMAR32to63	Numeric	0x0
C64PLUSMAR64to95	Numeric	0x0
C64PLUSMAR96to127	Numeric	0x0
C64PLUSMAR128to159	Numeric	0x0
C64PLUSMAR160to191	Numeric	0x0
C64PLUSMAR192to223	Numeric	0x0
C64PLUSMAR224to255	Numeric	0x0

Name	Type	Default (Enum Options)
GEMTRUECOMPEN	Bool	false (OMAP 2430/3430 only)
BCACHEREADADDR0	Numeric	0x0 (OMAP 2430/3430 only)
BCACHEREADADDR1	Numeric	0x0 (OMAP 2430/3430 only)
BCACHEREADADDR2	Numeric	0x0 (OMAP 2430/3430 only)

### Description

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

## GBL Module Properties

The following Global Settings can be made:

- ❑ **Target Board Name.** The name of the board or board family.  
Tconf Name: BOARDNAME Type: String  
Example: `bios.GBL.BOARDNAME = "c6xxx";`
- ❑ **Processor ID (PROCID).** ID used to communicate with other processors using the MSGQ Module. The proclid is also defined in the MSGQ\_TransportObj array that is part of the MSGQ\_Config structure. This value can be obtained with GBL\_getProclid and modified by GBL\_setProclid (but only within the User Init Function).  
Tconf Name: PROCID Type: Int16  
Example: `bios.GBL.PROCID = 0;`
- ❑ **Board Clock In KHz (Informational Only).** Frequency of the input clock in KHz. You should set this property to match the actual board clock rate. This property does not change the rate of the board; it is informational only. The configured value can be obtained at run-time using the GBL\_getClkin API. The default value is 20000 KHz.  
Tconf Name: CLKIN Type: Uint32  
Example: `bios.GBL.CLKIN = 20000;`
- ❑ **DSP Speed In MHz (CLKOUT).** This number, times 1000000, is the number of instructions the processor can execute in 1 second. You should set this property to match the actual rate. This property does not change the rate of the board. This value is used by the CLK manager to calculate register settings for the on-device timers.  
Tconf Name: CLKOUT Type: Int16  
Example: `bios.GBL.CLKOUT = 133.0000;`
- ❑ **Specify RTS Library.** Determines whether a user can specify the run-time support library to which the application is linked. The RTS library contains the printf, malloc, and other standard C library



functions. For information about using this library, see “std.h and stdlib.h functions” on page 2-562. If you do not choose to specify a library, the default library for your platform is used.

Tconf Name: SPECIFYRTSLIB Type: Bool

Example: `bios.GBL.SPECIFYRTSLIB = false;`

- ❑ **Run-Time Support Library.** The name of the run-time support (RTS) library to which the application is linked. These libraries are located in the <BIOS\_INSTALL\_DIR>\xdctools\packages\ti\targets tree. The library you select is used in the linker command file generated from the Tconf script when you build your application.

Tconf Name: RTSLIB Type: String

Example: `bios.GBL.RTSLIB = "";`

- ❑ **DSP Endian Mode.** 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. This property must match the setting in the DSP's CSR register.

Tconf Name: ENDIANMODE Type: EnumString

Options: "little", "big"

Example: `bios.GBL.ENDIANMODE = "little";`

- ❑ **Call User Init Function.** Set this property to true if you want an initialization function to be called early during program initialization, after .cinit processing and before the main() function.

Tconf Name: CALLUSERINITFXN Type: Bool

Example: `bios.GBL.CALLUSERINITFXN = false;`

- ❑ **User Init Function.** Type the name of the initialization function. This function runs early in the initialization process and is intended to be used to perform hardware setup that needs to run before DSP/BIOS is initialized. The code in this function should not use any DSP/BIOS API calls, unless otherwise specified for that API, since a number of DSP/BIOS modules have not been initialized when this function runs. In contrast, the Initialization function that may be specified for HOOK Module objects runs later and is intended for use in setting up data structures used by other functions of the same HOOK object.

Tconf Name: USERINITFXN Type: Extern

Example: `bios.GBL.USERINITFXN =  
prog.extern("FXN_F_nop");`

- ❑ **Enable Real Time Analysis.** If this property is true, target-to-host communication is enabled by the addition of IDL objects to run the IDL\_cpuLoad, LNK\_dataPump, and RTA\_dispatch functions. If this property is false, these IDL objects are removed and target-to-host communications are not supported. As a result, support for DSP/BIOS implicit instrumentation is removed.

Tconf Name: ENABLEINST

Type: Bool

Example: `bios.GBL.ENABLEINST = true;`

- ❑ **Use Instrumented BIOS Library.** Specifies whether to link with the instrumented or non-instrumented version of the DSP/BIOS library. The non-instrumented versions are somewhat smaller but do not provide support for LOG, STS, and TRC instrumentation. The libraries are located in <BIOS\_INSTALL\_DIR>\packages\ti\bios\lib. By default, the instrumented version of the library for your platform is used.

Tconf Name: INSTRUMENTED

Type: Bool

Example: `bios.GBL.INSTRUMENTED = true;`

- ❑ **Enable All TRC Trace Event Classes.** Set this property to false 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.

Tconf Name: ENABLEALLTRC

Type: Bool

Example: `bios.GBL.ENABLEALLTRC = true;`

- ❑ **Program Cache Control - CSR(PCC).** This property in the DSP family tab specifies the cache mode for the DSP at program initiation.

Tconf Name: CSR PCC

Type: EnumString

Options: "mapped", "cache enable", "cache freeze", "cache bypass"

Example: `bios.GBL.CSRPCC = "mapped";`

## 621x/671x tab

- ❑ **Configure L2 Memory Settings.** You can set this property to true for DSPs that have a L1/L2 cache (for example, the c6211). The other L2 properties on this tab are available if this property is true.

Tconf Name: C621XCONFIGUREL2

Type: Bool

Example: `bios.GBL.C621XCONFIGUREL2 = false;`

- ❑ **L2 Mode - CCFG(L2MODE).** (621x/671x and 641x tabs) Sets the L2 cache mode. See the *c6000 Peripherals Manual* for details.

Tconf Name: C621XCCFGL2MODE                      Type: EnumString

Options:        "SRAM", "1-way cache", "2-way cache", "3-way cache", "4-way cache"

Example:        `bios.GBL.C621XCCFGL2MODE =  
                  "4-way cache (0k)";`

- ❑ **MAR 0-15 - bitmask used to initialize MARs.** Only bit 0 of each of these 32-bit registers is modifiable by the user. All other bits are reserved. Specify a bitmask for the 16 modifiable bits in registers MAR0 through MAR15. The lowest bit of the bitmask you specify corresponds to the smallest MAR number in this range. That is, bit 0 corresponds to the 0 bit of MAR0 and bit 15 corresponds to the 0 bit of MAR15.

Tconf Name: C621XMAR                                      Type: Numeric

Example:        `bios.GBL.C621XMAR = 0x0000;`

## 641x tab

- ❑ **Configure L2 Memory Settings.** You can set this property to true for DSPs that have a L1/L2 cache (for example, the c6211). The other L2 properties on this tab are available if this property is true.

Tconf Name: C641XCONFIGUREL2                      Type: Bool

Example:        `bios.GBL.C641XCONFIGUREL2 = false;`

- ❑ **L2 Mode - CCFG(L2MODE).** Sets the L2 cache mode. See the *c6000 Peripherals Manual* for details.

Tconf Name: C641XCCFGL2MODE                      Type: EnumString

Options:        "4-way cache (0k)", "4-way cache (32k)",  
                  "4-way cache (64k)", "4-way cache (128k)", "4-way  
                  cache (256k)"

Example:        `bios.GBL.C641XCCFGL2MODE =  
                  "4-way cache (0k)";`

- ❑ **MAR96-101 - bitmask controls EMIFB CE space.**  
**MAR128-143 - bitmask controls EMIFA CE0 space.**  
**MAR144-159 - bitmask controls EMIFA CE1 space.**  
**MAR160-175 - bitmask controls EMIFA CE2 space.**  
**MAR176-191 - bitmask controls EMIFA CE3 space.**  
 Only bit 0 of each of these 32-bit registers is modifiable by the user. All other bits are reserved. Specify a bitmask for the modifiable bits in registers MAR96 through MAR101. The lowest bit of the bitmask

you specify corresponds to the smallest MAR number in this range. For example, in C641XMARCE0, bit 0 corresponds to the 0 bit of MAR128 and bit 15 corresponds to the 0 bit of MAR143.

Tconf Name: C641XMAREMIFB Type: Numeric

Tconf Name: C641XMARCE0 Type: Numeric

Tconf Name: C641XMARCE1 Type: Numeric

Tconf Name: C641XMARCE2 Type: Numeric

Tconf Name: C641XMARCE3 Type: Numeric

Example: `bios.GBL.C641XMAREMIFB = 0x0000;`

- ❑ **L2 Requestor Priority - CCFG(P).** Specifies the CPU/DMA cache priority. See the *c6000 Peripherals Manual* for details.

Tconf Name: C641XCCFGP Type: EnumString

Options: "urgent", "high", "medium", "low"

Example: `bios.GBL.C641XCCFGP = "urgent";`

- ❑ **Configure Priority Queues.** Set this property to true if you want to configure the maximum number of transfer requests on the L2 priority queues.

Tconf Name: C641XSETL2ALLOC Type: Bool

Example: `bios.GBL.C641XSETL2ALLOC = false;`

- ❑ **Max L2 Transfer Requests on URGENT Queue (L2ALLOC0).** Select a number from 0 to 7 for the maximum number of L2 transfer requests permitted on the URGENT queue.

Tconf Name: C641XL2ALLOC0 Type: EnumInt

Options: 0 to 7

Example: `bios.GBL.C641XL2ALLOC0 = 6;`

- ❑ **Max L2 Transfer Requests on HIGH Queue (L2ALLOC1).** Select a number from 0 to 7 for the maximum number of L2 transfer requests permitted on the HIGH priority queue.

Tconf Name: C641XL2ALLOC1 Type: EnumInt

Options: 0 to 7

Example: `bios.GBL.C641XL2ALLOC1 = 2;`

- ❑ **Max L2 Transfer Requests on MEDIUM Queue (L2ALLOC2).** Select a number from 0 to 7 for the maximum number of L2 transfer requests permitted on the MEDIUM priority queue.

Tconf Name: C641XL2ALLOC2 Type: EnumInt

Options: 0 to 7

Example: `bios.GBL.C641XL2ALLOC2 = 2;`

**64PLUS tab**

- ☐ **Max L2 Transfer Requests on LOW Queue (L2ALLOC3).** Select a number from 0 to 7 for the maximum number of L2 transfer requests permitted on the LOW priority queue.

Tconf Name: C641XL2ALLOC3

Type: EnumInt

Options: 0 to 7

Example: `bios.GBL.C641XL2ALLOC3 = 2;`

- ☐ **64P - Configure Memory Cache Settings.** You can set this property to true if you want to configure the cache settings for the 'C64x+ initialization. Checking this box enables the cache size and MAR bitmask properties that follow on this tab.

Tconf Name: C64PLUSCONFIGURE

Type: Bool

Example: `bios.GBL.C64PLUSCONFIGURE = false;`

- ☐ **64P L1PCFG Mode.** Select the initial size for the L1P cache. See the *c6000 Peripherals Manual* for details.

Tconf Name: C64PLUSL1PCFG

Type: EnumString

Options: "0k", "4k", "8k", "16k", "32k"

Example: `bios.GBL.C64PLUSL1PCFG = "32k";`

- ☐ **64P L1DCFG Mode.** Select the initial size for the L1D cache.

Tconf Name: C64PLUSL1DCFG

Type: EnumString

Options: "0k", "4k", "8k", "16k", "32k"

Example: `bios.GBL.C64PLUSL1DCFG = "32k";`

- ☐ **64P L2CFG Mode.** Select the initial size for the L2 cache.

Tconf Name: C64PLUSL2CFG

Type: EnumString

Options: "0k", "32k", "64k", "128k", "256k"

Example: `bios.GBL.C64PLUSL2CFG = "32k";`

- ☐ **MAR - bitmasks.** Only bit 0 of each of these 32-bit registers is modifiable by the user. All other bits are reserved. Specify a bitmask for the 32 modifiable bits in the registers specified for the property. The lowest bit of the bitmask you specify corresponds to the smallest

MAR number in this range. For example, in C64PLUSMAR128to159, bit 0 corresponds to the 0 bit of MAR128 and bit 31 corresponds to the 0 bit of MAR159.

Tconf Name: C64PLUSMAR0to31 Type: Numeric

Tconf Name: C64PLUSMAR32to63 Type: Numeric

Tconf Name: C64PLUSMAR64to95 Type: Numeric

Tconf Name: C64PLUSMAR96to127 Type: Numeric

Tconf Name: C64PLUSMAR128to159 Type: Numeric

Tconf Name: C64PLUSMAR160to191 Type: Numeric

Tconf Name: C64PLUSMAR192to223 Type: Numeric

Tconf Name: C64PLUSMAR224to255 Type: Numeric

Example: `bios.GBL.C64PLUSMAR0to31 = 0x0;`

- ☐ **GEM True Completion Bit.** Set this property to true to enable the GEM True Completion bit. This controls how cache writeback completion works. See the OMAP2430/3430 cache documentation for more information on how to ensure that a cache writeback is complete. Checking this box enables the BCACHE read address properties that follow. (OMAP 2430/3430 only)

Tconf Name: GEMTRUECOMPEN Type: Bool

Example: `bios.GBL.GEMTRUECOMPEN = false;`

- ☐ **BCACHE Read Address 0-2.** Specify the first, second, and third addresses to read back during BCACHE\_wait. Reading a non-cached address is necessary to ensure that a writeback has fully completed. DSP/BIOS provides properties to specify up to three "read addresses" because there are three memory regions for which you may want to ensure that writebacks are fully complete: SDRAM, flash, and OCM (on chip memory). For example, you can specify an address in SDRAM and an address in OCM for the first two properties and leave the last property at 0x0 if you do not use flash memory. The addresses are not read if the value is zero. See the OMAP2430/3430 cache documentation for more information. (OMAP 2430/3430 only)

Tconf Name: BCACHEREADADDR0 Type: Numeric

Tconf Name: BCACHEREADADDR1 Type: Numeric

Tconf Name: BCACHEREADADDR2 Type: Numeric

Example: `bios.GBL.BCACHEREADADDR0 = 0x0;`

**GBL\_getClkin***Get configured value of board input clock in KHz***C Interface****Syntax**                      `clkin = GBL_getClkin(Void);`**Parameters**                  `Void`**Return Value**                `UInt32            clkin;        /* CLKIN frequency */`**Reentrant**                    `yes`**Description**                Returns the configured value of the board input clock (CLKIN) frequency in KHz.**See Also**                    `CLK_countspms`  
`CLK_getprd`

**GBL\_getFrequency***Get current frequency of the CPU in KHz***C Interface****Syntax** frequency = GBL\_getFrequency(Void);**Parameters** Void**Return Value** Uint32 frequency; /\* CPU frequency in KHz \*/**Reentrant** yes**Description** Returns the current frequency of the DSP CPU in an integer number of KHz. This is the frequency set by GBL\_setFrequency, which must also be an integer. The default value is the value of the CLKOUT property, which is configured as one of the GBL Module Properties.**See Also** GBL\_getClkin  
GBL\_setFrequency



**GBL\_getProcid***Get configured value of processor ID***C Interface**

**Syntax**                      `procid = GBL_getProcid(Void);`

**Parameters**                `Void`

**Return Value**              `UInt16              procid;    /* processor ID */`

**Reentrant**                      `yes`

**Description**                      Returns the configured value of the processor ID (PROCID) for this processor. This numeric ID value is used by the MSGQ module when determining which processor to communicate with.

The `procid` is also defined in the `MSGQ_TransportObj` array that is part of the `MSGQ_Config` structure. The same processor ID should be defined for this processor in both locations.

During the User Init Function, the application may modify the statically configured processor ID by calling `GBL_setProcid`. In this case, the User Init Function may need to call `GBL_getProcid` first to get the statically configured processor ID.

**See Also**                      MSGQ Module: Static Configuration  
                                 `GBL_setProcid`

GBL\_getVersion

Get DSP/BIOS version information

C Interface

Syntax	version = GBL_getVersion(Void);
Parameters	Void
Return Value	UInt16          version;    /* version data */
Reentrant	yes
Description	Returns DSP/BIOS kernel version information as a 4-digit hex number. For example: 0x5100. Note that the kernel version is different from the DSP/BIOS product version.

When comparing versions, compare the highest digits that are different. The digits in the version information are as follows:

Bits	Compatibility with Older DSP/BIOS Versions
12-15 (first hex digit)	Not compatible. Changes to application C, assembly, or configuration (Tconf) code may be required. For example, moving from 0x5100 to 0x6100 may require code changes.
8-11 (second hex digit)	No code changes required but you should recompile. For example, moving from 0x5100 to 0x5200 requires recompilation.
0-7 (third and fourth hex digits)	No code changes or recompile required. You should re-link if either of these digits are different. For example, moving from 0x5100 to 0x5102 requires re-linking.

The version returned by GBL\_getVersion matches the version in the DSP/BIOS header files. (For example, tsk.h.) If the header file version is as follows, GBL\_getVersion returns 0x5001. If there are three items, the last item uses two digits (for example, 01) in the returned hex number.

\*    @(#) DSP/BIOS\_Kernel 5,0,1 05-30-2004 (cuda-106)

**GBL\_setFrequency***Set frequency of the CPU in KHz***C Interface****Syntax**

GBL\_setFrequency( frequency );

**Parameters**

UInt32 frequency; /\* CPU frequency in KHz \*/

**Return Value**

Void

**Reentrant**

yes

**Description**

This function sets the value of the CPU frequency known to DSP/BIOS.

Note that GBL\_setFrequency does not affect the PLL, and therefore has no effect on the actual frequency at which the DSP is running. It is used only to make DSP/BIOS aware of the DSP frequency you are using.

If you call GBL\_setFrequency to update the CPU frequency known to DSP/BIOS, you should follow the sequence shown in the CLK\_reconfig topic to reconfigure the timer.

The frequency must be an integer number of KHz.

**Constraints and Calling Context**

- ❑ If you change the frequency known to DSP/BIOS, you should also reconfigure the timer (with CLK\_reconfig) so that the actual frequency is the same as the frequency known to DSP/BIOS.

**See Also**

CLK\_reconfig  
 GBL\_getClkin  
 GBL\_getFrequency

**GBL\_setProclD***Set configured value of processor ID***C Interface****Syntax**

```
GBL_setProclD( proclD );
```

**Parameters**

```
UInt16      proclD;      /* processor ID */
```

**Return Value**

```
Void
```

**Reentrant**

```
no
```

**Description**

Sets the processor ID (PROCID) for this processor. This numeric ID value is used by the MSGQ module to determine which processor to communicate with.

The proclD is also defined in the MSGQ\_TransportObj array that is part of the MSGQ\_Config structure.

This function can only be called in the User Init Function configured as part of the GBL Module Properties. That is, this function may only be called at the beginning of DSP/BIOS initialization.

The application may determine the true processor ID for the device during the User Init Function and call GBL\_setProclD with the correct processor ID. This is useful in applications that run a single binary image on multiple DSP processors.

How the application determines the correct processor ID is application- or board-specific. For example, you might use GPIO. You can call GBL\_getProclD from the User Init Function to get the statically configured processor ID.

**Constraints and Calling Context**

- ❑ This function can only be called in the User Init Function configured as part of the GBL Module Properties.

**See Also**

MSGQ Manager Properties  
GBL\_getProclD

## 2.9 GIO Module

The GIO module is the Input/Output Module used with IOM mini-drivers as described in *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

### Functions

- ❑ `GIO_abort`. Abort all pending input and output.
- ❑ `GIO_control`. Device specific control call.
- ❑ `GIO_create`. Allocate and initialize a GIO object.
- ❑ `GIO_delete`. Delete underlying mini-drivers and free up the GIO object and any associated IOM packet structures.
- ❑ `GIO_flush`. Drain output buffers and discard any pending input.
- ❑ `GIO_new`. Initialize a GIO object using pre-allocated memory.
- ❑ `GIO_read`. Synchronous read command.
- ❑ `GIO_submit`. Submits a packet to the mini-driver.
- ❑ `GIO_write`. Synchronous write command.

### Constants, Types, and Structures

```
/* Modes for GIO_create */
#define IOM_INPUT      0x0001
#define IOM_OUTPUT     0x0002
#define IOM_INOUT      (IOM_INPUT | IOM_OUTPUT)

/* IOM Status and Error Codes */
#define IOM_COMPLETED SYS_OK /* I/O successful */
#define IOM_PENDING     1 /* I/O queued and pending */
#define IOM_FLUSHED     2 /* I/O request flushed */
#define IOM_ABORTED     3 /* I/O aborted */
#define IOM_EBADIO      -1 /* generic failure */
#define IOM_ETIMEOUT    -2 /* timeout occurred */
#define IOM_ENOPACKETS  -3 /* no packets available */
#define IOM_EFREE       -4 /* unable to free resources */
#define IOM_EALLOC      -5 /* unable to alloc resource */
#define IOM_EABORT      -6 /* I/O aborted uncompleted */
#define IOM_EBADMODE    -7 /* illegal device mode */
#define IOM_EOF         -8 /* end-of-file encountered */
#define IOM_ENOTIMPL    -9 /* operation not supported */
#define IOM_EBADARGS    -10 /* illegal arguments used */
#define IOM_ETIMEOUTUNREC -11
                        /* unrecoverable timeout occurred */
#define IOM_EINUSE      -12 /* device already in use */
```

```

/* Command codes for IOM_Packet */
#define IOM_READ    0
#define IOM_WRITE   1
#define IOM_ABORT   2
#define IOM_FLUSH   3
#define IOM_USER     128 /* 0-127 reserved for system */

/* Command codes reserved for control */
#define IOM_CHAN_RESET    0 /* reset channel only */
#define IOM_CHAN_TIMEOUT 1
                        /* channel timeout occurred */
#define IOM_DEVICE_RESET  2 /* reset entire device */
#define IOM_CNTL_USER     128
                        /* 0-127 reserved for system */

/* Structure passed to GIO_create */
typedef struct GIO_Attrs {
    Int    nPackets; /* number of asynch I/O packets */
    Uns    timeout;  /* for blocking (SYS_FOREVER) */
} GIO_Attrs;

/* Struct passed to GIO_submit for synchronous use*/
typedef struct GIO_AppCallback {
    GIO_TappCallback    fxn;
    Ptr                 arg;
} GIO_AppCallback;

typedef struct GIO_Obj {
    IOM_Fxns    *fxns; /* ptr to function table */
    Uns         mode;  /* create mode */
    Uns         timeout; /* timeout for blocking */
    IOM_Packet  syncPacket; /* for synchronous use */
    QUE_Obj     freeList; /* frames for asynch I/O */
    Ptr         syncObj;  /* ptr to synchro. obj */
    Ptr         mdChan;   /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;

typedef struct IOM_Fxns
{
    IOM_TmdBindDev      mdBindDev;
    IOM_TmdUnBindDev    mdUnBindDev;
    IOM_TmdControlChan  mdControlChan;
    IOM_TmdCreateChan   mdCreateChan;
    IOM_TmdDeleteChan   mdDeleteChan;
    IOM_TmdSubmitChan   mdSubmitChan;
} IOM_Fxns;

```

```
typedef struct IOM_Packet { /* frame object */
    QUE_Elem link; /* queue link */
    Ptr      addr; /* buffer address */
    size_t   size; /* buffer size */
    Arg      misc; /* reserved for driver */
    Arg      arg; /* user argument */
    Uns      cmd; /* mini-driver command */
    Int      status; /* status of command */
} IOM_Packet;
```

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the GIO Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

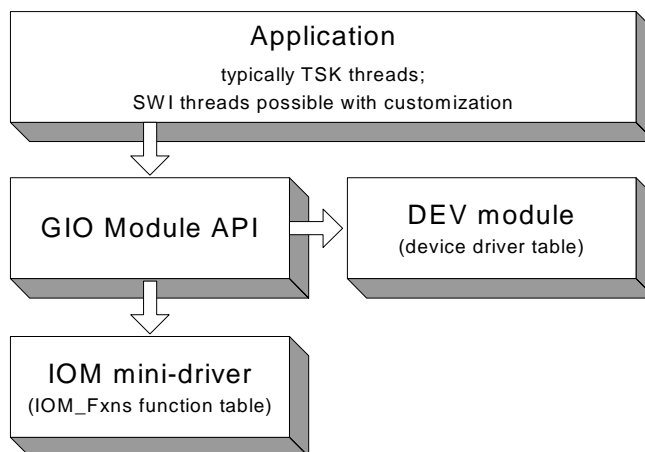
Name	Type	Default
ENABLEGIO	Bool	false
CREATEFXN	Extern	prog.extern("FXN_F_nop")
DELETFXN	Extern	prog.extern("FXN_F_nop")
PENDFXN	Extern	prog.extern("FXN_F_nop")
POSTFXN	Extern	prog.extern("FXN_F_nop")

**Description**

The GIO module provides a standard interface to mini-drivers for devices such as UARTs, codecs, and video capture/display devices. The creation of such mini-drivers is not covered in this manual; it is described in *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

The GIO module is independent of the actual mini-driver being used. It allows the application to use a common interface for I/O requests. It also handles response synchronization. It is intended as common "glue" to bind applications to device drivers.

The following figure shows how modules are related in an application that uses the GIO module and an IOM mini-driver:



The GIO module is the basis of communication between applications and mini-drivers. The DEV module is responsible for maintaining the table of device drivers that are present in the system. The GIO module obtains device information by using functions such as DEV\_match.

## GIO Manager Properties

The following global properties can be set for the GIO module in the GIO Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Enable General Input/Output Manager.** Set this property to true to enable use of the GIO module. If your application does not use GIO, you should leave it disabled to prevent additional modules (such as SEM) from being linked into your application.

Tconf Name: ENABLEGIO

Type: Bool

Example: `bios.GIO.ENABLEGIO = false;`

- ❑ **Create Function.** The function the GIO module should use to create a synchronization object. This function is typically SEM\_create. If you use another function, that function should have a prototype that matches that of SEM\_create: `Ptr CREATEFXN(Int count, Ptr attrs);`

Tconf Name: CREATEFXN

Type: Extern

Example: `bios.GIO.CREATEFXN =  
          prog.extern("SEM_create");`



- ❑ **Delete Function.** The function the GIO module should use to delete a synchronization object. This function is typically `SEM_delete`. If you use another function, that function should have a prototype that matches that of `SEM_delete`: `Void DELETEFXN(Ptr semHandle);`

Tconf Name: `DELETEFXN`

Type: Extern

Example: `bios.GIO.DELETEFXN =  
          prog.extern("SEM_delete");`

- ❑ **Pend Function.** The function the GIO module should use to pend on a synchronization object. This function is typically `SEM_pend`. If you use another function, that function should have a prototype that matches that of `SEM_pend`: `Bool PENDFXN(Ptr semHandle, Uns timeout);`

Tconf Name: `PENDFXN`

Type: Extern

Example: `bios.GIO.PENDFXN =  
          prog.extern("SEM_pend");`

- ❑ **Post Function.** The function the GIO module should use to post a synchronization object. This function is typically `SEM_post`. If you use another function, that function should have a prototype that matches that of `SEM_post`: `Void POSTFXN(Ptr semHandle);`

Tconf Name: `POSTFXN`

Type: Extern

Example: `bios.GIO.POSTFXN =  
          prog.extern("SEM_post");`

## GIO Object Properties

GIO objects cannot be created statically. In order to create a GIO object, the application should call `GIO_create` or `GIO_new`.

**GIO\_abort***Abort all pending input and output***C Interface**

**Syntax** `status = GIO_abort(gioChan);`

**Parameters** `GIO_Handle gioChan; /* handle to an instance of the device */`

**Return Value** `Int status; /* returns IOM_COMPLETED if successful */`

**Description**

An application calls `GIO_abort` to abort all input and output from the device. When this call is made, all pending calls are completed with a status of `GIO_ABORTED`. An application uses this call to return the device to its initial state. Usually this is done in response to an unrecoverable error at the device level.

`GIO_abort` returns `IOM_COMPLETED` upon successfully aborting all input and output requests. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_abort` results in a call to the `mdSubmit` function of the associated mini-driver. The `IOM_ABORT` command is passed to the `mdSubmit` function. The `mdSubmit` call is typically a blocking call, so calling `GIO_abort` can result in the thread blocking.

**Constraints and Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to `GIO_create` or `GIO_new`.
- ❑ `GIO_abort` cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

**Example**

```
/* abort all I/O requests given to the device*/
gioStatus = GIO_abort(gioChan);
```

**GIO\_control***Device specific control call***C Interface****Syntax**

```
status = GIO_control(gioChan, cmd, args);
```

**Parameters**

```
GIO_Handle  gioChan; /* handle to an instance of the device */
Int          cmd;     /* control functionality to perform */
Ptr          args;    /* data structure to pass control information */
```

**Return Value**

```
Int          status; /* returns IOM_COMPLETED if successful */
```

**Description**

An application calls `GIO_control` to configure or perform control functionality on the communication channel.

The `cmd` parameter may be one of the command code constants listed in “Constants, Types, and Structures” on page 2-143. A mini-driver may add command codes for additional functionality.

The `args` parameter points to a data structure defined by the device to allow control information to be passed between the device and the application. This structure can be generic across a domain or specific to a mini-driver. In some cases, this argument may point directly to a buffer holding control data. In other cases, there may be a level of indirection if the mini-driver expects a data structure to package many components of data required for the control operation. In the simple case where no data is required, this parameter may just be a predefined command value.

`GIO_control` returns `IOM_COMPLETED` upon success. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_control` results in a call to the `mdControl` function of the associated mini-driver. The `mdControl` call is typically a blocking call, so calling `GIO_control` can result in blocking.

**Constraints and Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to `GIO_create` or `GIO_new`.
- ❑ `GIO_control` cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

**Example**

```
/* Carry out control/configuration on the device*/
gioStatus = GIO_control(gioChan, XXX_RESET, &args);
```

**GIO\_create***Allocate and initialize a GIO object***C Interface**

<b>Syntax</b>	<code>gioChan = GIO_create(name, mode, *status, chanParams, *attrs)</code>		
<b>Parameters</b>	String	name	<i>/* name of the device to open */</i>
	Int	mode	<i>/* mode in which the device is to be opened */</i>
	Int	*status	<i>/* address to place driver return status */</i>
	Ptr	chanParams	<i>/* optional */</i>
	GIO_Attrs	*attrs	<i>/* pointer to a GIO_Attrs structure */</i>
<b>Return Value</b>	GIO_Handle	gioChan;	<i>/* handle to an instance of the device */</i>

**Description**

An application calls `GIO_create` to create a `GIO_Obj` object and open a communication channel. This function initializes the I/O channel and opens the lower-level device driver channel. The `GIO_create` call also creates the synchronization objects it uses and stores them in the `GIO_Obj` object.

The name argument is the name specified for the device when it was created in the configuration or at runtime.

The mode argument specifies the mode in which the device is to be opened. This may be `IOM_INPUT`, `IOM_OUTPUT`, or `IOM_INOUT`.

If the status returned by the device is non-NULL, a status value is placed at the address specified by the status parameter.

The `chanParams` parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner.

The `attrs` parameter is a pointer to a structure of type `GIO_Attrs`.

```
typedef struct GIO_Attrs {  
    Int  nPackets; /* number of asynch I/O packets */  
    Uns  timeout;  /* for blocking calls (SYS_FOREVER) */  
} GIO_Attrs;
```

If `attrs` is NULL, a default set of attributes is used. The default for `nPackets` is 2. The default for `timeout` is `SYS_FOREVER`.

The `GIO_create` call allocates a list of `IOM_Packet` items as specified by the `nPackets` member of the `GIO_Attrs` structure and stores them in the `GIO_Obj` object it creates.

GIO\_create returns a handle to the GIO\_Obj object created upon a successful open. The handle returned by this call should be used by the application in subsequent calls to GIO functions. This function returns a NULL handle if the device could not be opened. For example, if a device is opened in a mode not supported by the device, this call returns a NULL handle.

A call to GIO\_create results in a call to the mdCreateChan function of the associated mini-driver.

### Constraints and Calling Context

- ❑ A GIO stream can only be used by one task simultaneously. Catastrophic failure can result if more than one task calls GIO\_read on the same input stream, or more than one task calls GIO\_write on the same output stream.
- ❑ GIO\_create cannot be called from the context of a SWI or HWI thread.
- ❑ This function can be called only after the device has been loaded and initialized.

### Example

```
/* Create a device instance */
gioAttrs = GIO_ATTRS;
gioChan = GIO_create("\Codec0", IOM_INPUT, NULL, NULL,
                    &gioAttrs);
GIO_new
```

**GIO\_delete***Delete underlying mini-drivers and free GIO object and its structures***C Interface**

**Syntax** `status = GIO_delete(gioChan);`

**Parameters** `GIO_Handle gioChan; /* handle to device instance to be closed */`

**Return Value** `Int status; /* returns IOM_COMPLETED if successful */`

**Description**

An application calls `GIO_delete` to close a communication channel opened prior to this call with `GIO_create`. This function deallocates all memory allocated for this channel and closes the underlying device. All pending input and output are cancelled and the corresponding interrupts are disabled.

The `gioChan` parameter is the handle returned by `GIO_create` or `GIO_new`.

This function returns `IOM_COMPLETED` if the channel is successfully closed. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_delete` results in a call to the `mdDelete` function of the associated mini-driver.

**Constraints and Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to `GIO_create` or `GIO_new`.

**Example**

```
/* close the device instance */
GIO_delete(gioChan);
```

**GIO\_flush***Drain output buffers and discard any pending input***C Interface**

<b>Syntax</b>	<code>status = GIO_flush(gioChan);</code>
<b>Parameters</b>	<code>GIO_Handle gioChan; /* handle to an instance of the device */</code>
<b>Return Value</b>	<code>Int status; /* returns IOM_COMPLETED if successful */</code>

**Description**

An application calls `GIO_flush` to flush the input and output channels of the device. All input data is discarded; all pending output requests are completed. When this call is made, all pending input calls are completed with a status of `IOM_FLUSHED`, and all output calls are completed routinely.

The `gioChan` parameter is the handle returned by `GIO_create` or `GIO_new`.

This call returns `IOM_COMPLETED` upon successfully flushing all input and output. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_flush` results in a call to the `mdSubmit` function of the associated mini-driver. The `IOM_FLUSH` command is passed to the `mdSubmit` function. The `mdSubmit` call is typically a blocking call, so calling `GIO_flush` can result in the thread blocking while waiting for output calls to be completed.

**Constraints and Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to `GIO_create` or `GIO_new`.
- ❑ `GIO_flush` cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

**Example**

```
/* Flush all I/O given to the device*/
GIO_flush(gioChan);
```

**GIO\_new***Initialize a GIO object with pre-allocated memory***C Interface****Syntax**

```
gioChan = GIO_new(gioChan, name, mode, *status, optArgs,
packetBuf[], syncObject, *attrs);
```

**Parameters**

GIO_Handle	gioChan	/* Handle to GIO Obj */
String	name	/* name of the device to open */
Int	mode	/* mode in which the device is to be opened */
Int	*status	/* address to place driver return status */
Ptr	optArgs	/* optional args to mdCreateChan */
IOM_packet	packetBuf[]	/* to be initialized to zero */
Ptr	syncObject	/* sync Object */
GIO_Attrs	*attrs	/* pointer to a GIO_Attrs structure */

**Return Value**

```
GIO_Handle gioChan; /* handle to the initialized GIO object */
```

**Description**

An application calls `GIO_new` to initialize a `GIO_Obj` object and open a communication channel. This function initializes the I/O channel and opens the lower-level device driver channel. The `GIO_new` call *does not* allocate any memory. It requires pre-allocated memory.

The "gioChan" parameter is a handle to a structure of type `GIO_Obj` that your program has declared. `GIO_new` initializes this structure.

```
typedef struct GIO_Obj {
    IOM_Fxns    *fxns;      /* ptr to function table */
    Uns         mode;       /* create mode */
    Uns         timeout;    /* timeout for blocking */
    IOM_Packet  syncPacket; /* for synchronous use */
    QUE_Obj     freeList;   /* frames for asynch I/O */
    Ptr         syncObj;    /* ptr to synchro. obj */
    Ptr         mdChan;     /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;
```

The "name" parameter is the name previously specified for the device. It is used to find a matching name in the device table.

The "mode" parameter specifies the mode in which the device is to be opened. This may be `IOM_INPUT`, `IOM_OUTPUT`, or `IOM_INOUT`.

If the status returned by the device is non-NULL, a status value is placed at the address specified by the "status" parameter.



The "optArgs" parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner.

Use the "packetBuf[]" array to pass a list of IOM\_Packet items. The number of items should match the nPackets member of the GIO\_Attrs structure passed to the "attrs" parameter. GIO\_new initializes these IOM\_Packet items.

The "syncObject" parameter is usually a SEM handle.

The "attrs" parameter is a pointer to a structure of type GIO\_Attrs.

```
typedef struct GIO_Attrs {
    Int nPackets; /* number of asynch I/O packets */
    Uns timeout; /* for blocking calls (SYS_FOREVER) */
} GIO_Attrs;
```

If attrs is NULL, a default set of attributes is used. The default for nPackets is 2. The default for timeout is SYS\_FOREVER. GIO\_new initializes the packets, but does not allocate them.

GIO\_new returns the non-NULL handle to the GIO\_Obj when initialization is successful. The handle returned by this call should be used by the application in subsequent calls to GIO functions. Usually, this is the same handle passed to GIO\_new. However, GIO\_new returns a NULL handle if the device could not be initialized. For example, if a device is opened in a mode not supported by the device, this call returns a NULL handle.

A call to GIO\_new results in a call to the mdCreateChan function of the associated mini-driver.

## Constraints and Calling Context

- ❑ This function can be called only after the device has been loaded and initialized.

## Example

```
/* Initialize a device object */
output = GIO_new(&outObj, "/printf", IOM_OUTPUT,
    &status, NULL, outPacketBuf, outSem, &attrs);
GIO_create
```

**GIO\_read***Synchronous read command***C Interface****Syntax**

```
status = GIO_read(gioChan, bufp, *pSize);
```

**Parameters**

```
GIO_Handle  gioChan; /* handle to an instance of the device */  
Ptr         bufp     /* pointer to data structure for buffer data */  
size_t      *pSize   /* pointer to size of bufp structure */
```

**Return Value**

```
Int         status; /* returns IOM_COMPLETED if successful */
```

**Description**

An application calls `GIO_read` to read a specified number of MADUs (minimum addressable data units) from the communication channel.

The `gioChan` parameter is the handle returned by `GIO_create` or `GIO_new`.

The `bufp` parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the read data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data. For example, for video capture devices this structure may contain pointers to RGB buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The `pSize` parameter points to the size of the buffer or data structure pointed to by the `bufp` parameter. When the function returns, this parameter points to the number of MADUs read from the device. This parameter is relevant only if the `bufp` parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

`GIO_read` returns `IOM_COMPLETED` upon successfully reading the requested number of MADUs from the device. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_read` results in a call to the `mdSubmit` function of the associated mini-driver. The `IOM_READ` command is passed to the `mdSubmit` function. The `mdSubmit` call is typically a blocking call, so calling `GIO_read` can result in the thread blocking.

**Constraints and  
Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO\_create or GIO\_new.
- ❑ GIO\_read cannot be called from a SWI, HWI, or main() unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

**Example**

```
/* Read from the device */  
size = sizeof(readStruct);  
status = GIO_read(gioChan, &readStruct, &size);
```

**GIO\_submit***Submit a GIO packet to the mini-driver***C Interface****Syntax**

```
status = GIO_submit(gioChan, cmd, bufp, *pSize, *appCallback);
```

**Parameters**

```
GIO_Handle  gioChan; /* handle to an instance of the device */
Uns         cmd      /* specified mini-driver command */
Ptr         bufp     /* pointer to data structure for buffer data */
size_t      *pSize   /* pointer to size of bufp structure */
GIO_AppCallback *appCallback /* pointer to callback structure */
```

**Return Value**

```
Int         status; /* returns IOM_COMPLETED if successful */
```

**Description**

GIO\_submit is not typically called by applications. Instead, it is used internally and for user-defined extensions to the GIO module.

GIO\_read and GIO\_write are macros that call GIO\_submit with appCallback set to NULL. This causes GIO to complete the I/O request synchronously using its internal synchronization object (by default, a semaphore). If appCallback is non-NULL, the specified callback is called without blocking. This API is provided to extend GIO functionality for use with SWI threads without changing the GIO implementation.

The gioChan parameter is the handle returned by GIO\_create or GIO\_new.

The cmd parameter is one of the command code constants listed in “Constants, Types, and Structures” on page 2-143. A mini-driver may add command codes for additional functionality.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of MADUs transferred to or from the device. This parameter is relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

The `appCallback` parameter points to either a callback structure that contains the callback function to be called when the request completes, or it points to `NULL`, which causes the call to be synchronous. When a queued request is completed, the callback routine (if specified) is invoked (i.e. blocking).

`GIO_submit` returns `IOM_COMPLETED` upon successfully carrying out the requested functionality. If the request is queued, then a status of `IOM_PENDING` is returned. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_submit` results in a call to the `mdSubmit` function of the associated mini-driver. The specified command is passed to the `mdSubmit` function.

## Constraints and Calling Context

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to `GIO_create` or `GIO_new`.
- ❑ This function can be called within the program’s `main()` function only if the GIO channel is asynchronous (non-blocking).

## Example

```
/* write asynchronously to the device*/
size = sizeof(userStruct);
status = GIO_submit(gioChan, IOM_WRITE, &userStruct,
                   &size, &callbackStruct);

/* write synchronously to the device */
size = sizeof(userStruct);
status = GIO_submit(gioChan, IOM_WRITE, &userStruct,
                   &size, NULL);
```

**GIO\_write***Synchronous write command***C Interface****Syntax**

```
status = GIO_write(gioChan, bufp, *pSize);
```

**Parameters**

```
GIO_Handle  gioChan; /* handle to an instance of the device */  
Ptr         bufp     /* pointer to data structure for buffer data */  
size_t      *pSize   /* pointer to size of bufp structure */
```

**Return Value**

```
Int         status; /* returns IOM_COMPLETED if successful */
```

**Description**

The application uses this function to write a specified number of MADUs to the communication channel.

The `gioChan` parameter is the handle returned by `GIO_create` or `GIO_new`.

The `bufp` parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the write data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be written to, and other device-dependent data. For example, for video capture devices this structure may contain pointers to RGB buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The `pSize` parameter points to the size of the buffer or data structure pointed to by the `bufp` parameter. When the function returns, this parameter points to the number of MADUs written to the device. This parameter is relevant only if the `bufp` parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

`GIO_write` returns `IOM_COMPLETED` upon successfully writing the requested number of MADUs to the device. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 2-143.

A call to `GIO_write` results in a call to the `mdSubmit` function of the associated mini-driver. The `IOM_WRITE` command is passed to the `mdSubmit` function. The `mdSubmit` call is typically a blocking call, so calling `GIO_write` can result in blocking.

**Constraints and  
Calling Context**

- ❑ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO\_create or GIO\_new.
- ❑ This function can be called within the program's main() function only if the GIO channel is asynchronous (non-blocking).
- ❑ GIO\_write cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

**Example**

```
/* write synchronously to the device*/  
size = sizeof(writeStruct);  
status = GIO_write(gioChan, &writeStrct, &size);
```

## 2.10 HOOK Module

The HOOK module is the Hook Function manager.

### Functions

- ❑ `HOOK_getenv`. Get environment pointer for a given HOOK and TSK combination.
- ❑ `HOOK_setenv`. Set environment pointer for a given HOOK and TSK combination.

### Constants, Types, and Structures

```
typedef Int HOOK_Id;          /* HOOK instance id */

typedef Void (*HOOK_InitFxn)(HOOK_Id id);
typedef Void (*HOOK_CreateFxn)(TSK_Handle task);
typedef Void (*HOOK_DeleteFxn)(TSK_Handle task);
typedef Void (*HOOK_ExitFxn)(Void);
typedef Void (*HOOK_ReadyFxn)(TSK_Handle task);
typedef Void (*HOOK_SwitchFxn)(TSK_Handle prev,
                               TSK_Handle next);
```

### Configuration Properties

The following list shows the properties that can be configured in a `Tconf` script, along with their types and default values. For details, see the HOOK Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
initFxn	Extern	prog.extern("FXN_F_nop")
createFxn	Extern	prog.extern("FXN_F_nop")
deleteFxn	Extern	prog.extern("FXN_F_nop")
exitFxn	Extern	prog.extern("FXN_F_nop")
callSwitchFxn	Bool	false
switchFxn	Extern	prog.extern("FXN_F_nop")
callReadyFxn	Bool	false
readyFxn	Extern	prog.extern("FXN_F_nop")
order	Int16	2

### Description

The HOOK module is an extension to the TSK function hooks defined in the TSK Manager Properties. It allows multiple sets of hook functions to be performed at key execution points. For example, an application that integrates third-party software may need to perform both its own hook functions and the hook functions required by the third-party software.



In addition, each HOOK object can maintain private data environments for each task for use by its hook functions.

The key execution points at which hook functions can be executed are during program initialization and at several TSK execution points.

The HOOK module manages objects that reference a set of hook functions. Each HOOK object is assigned a numeric identifier during DSP/BIOS initialization. If your program calls HOOK API functions, you must implement an initialization function for the HOOK instance that records the identifier in a variable of type `HOOK_Id`. DSP/BIOS passes the HOOK object's ID to the initialization function as the lone parameter.

The following function, `myInit`, could be configured as the Initialization function for a HOOK object using `Tconf`.

```
#include <hook.h>
HOOK_Id myId;

Void myInit(HOOK_Id id)
{
    myId = id;
}
```

The `HOOK_setenv` function allows you to associate an environment pointer to any data structure with a particular HOOK object and TSK object combination.

There is no limit to the number of HOOK objects that can be created. However, each object requires a small amount of memory in the `.bss` section to contain the object.

A HOOK object initially has all of its functions set to `FXN_F_nop`. You can set some hook functions and use this no-op function for the remaining events. Since the switch and ready events occur frequently during real-time processing, a separate property controls whether any function is called.

When you create a HOOK object, any TSK module hook functions you have specified are automatically placed in a HOOK object called `HOOK_KNL`. To set any properties of this object other than the Initialization function, use the TSK module. To set the Initialization function property of the `HOOK_KNL` object, use the HOOK module.

When an event occurs, all HOOK functions for that event are called in the order set by the order property in the configuration. When you select the HOOK manager in the DSP/BIOS Configuration Tool, you can change the execution order by dragging objects within the ordered list.

## HOOK Manager Properties

There are no global properties for the HOOK manager. HOOK objects are placed in the C Variables Section (.bss).

## HOOK Object Properties

The following properties can be set for a HOOK object in the DPI Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script. To create a HOOK object in a configuration script, use the following syntax:

```
var myHook = bios.HOOK.create("myHook");
```

The Tconf examples that follow assume the object has been created as shown.

- ❑ **comment.** A comment to identify this HOOK object.

Tconf Name: comment

Type: String

Example: `myHook.comment = "HOOK funcs";`

- ❑ **Initialization function.** The name of a function to call during program initialization. Such functions run during the BIOS\_init portion of application startup, which runs before the program's main() function. Initialization functions can call most functions that can be called from the main() function. However, they should not call TSK module functions, because the TSK module is initialized after initialization functions run. In addition to code specific to the module hook, this function should be used to record the object's ID, if it is needed in a subsequent hook function. This initialization function is intended for use in setting up data structures used by other functions of the same HOOK object. In contrast, the User Init Function property of the GBL Module Properties runs early in the initialization process and is intended to be used to perform hardware setup that needs to run before DSP/BIOS is initialized.

Tconf Name: initFxn

Type: Extern

Example: `myHook.initFxn = prog.extern("myInit");`

- ❑ **Create function.** The name of a function to call when any task is created. This includes tasks that are created statically and those created dynamically using TSK\_create. The TSK\_create topic describes the prototype required for the Create function. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

Tconf Name: createFxn

Type: Extern

Example: `myHook.createFxn =  
          prog.extern("myCreate");`

- ❑ **Delete function.** The name of a function to call when any task is deleted at run-time with TSK\_delete.

Tconf Name: deleteFxn

Type: Extern

Example: `myHook.deleteFxn =  
          prog.extern("myDelete");`

- ❑ **Exit function.** The name of a function to call when any task exits. The TSK\_exit topic describes the Exit function.

Tconf Name: exitFxn

Type: Extern

Example: `myHook.exitFxn = prog.extern("myExit");`

- ❑ **Call switch function.** Set this property to true if you want a function to be called when any task switch occurs.

Tconf Name: callSwitchFxn

Type: Bool

Example: `myHook.callSwitchFxn = false;`

- ❑ **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. The TSK Module topic describes the Switch function.

Tconf Name: switchFxn

Type: Extern

Example: `myHook.switchFxn =  
          prog.extern("mySwitch");`

- ❑ **Call ready function.** Set this property to true if you want a function to be called when any task becomes ready to run.

Tconf Name: callReadyFxn

Type: Bool

Example: `myHook.callReadyFxn = false;`

- ❑ **Ready function.** The name of a function to call when any task becomes ready to run. The TSK Module topic describes the Ready function.

Tconf Name: readyFxn

Type: Extern

Example: `myHook.readyFxn =  
          prog.extern("myReady");`

- ❑ **order.** Set this property for all HOOK function objects match the order in which HOOK functions should be executed.

Tconf Name: order

Type: Int16

Example: `myHook.order = 2;`

**HOOK\_getenv***Get environment pointer for a given HOOK and TSK combination***C Interface**

**Syntax**                      `environ = HOOK_getenv(task, id);`

**Parameters**                `TSK_Handle task;        /* task object handle */`  
                              `HOOK_Id     id;        /* HOOK instance id */`

**Return Value**              `Ptr            environ; /* environment pointer */`

**Reentrant**                      `yes`

**Description**                HOOK\_getenv returns the environment pointer associated with the specified HOOK and TSK objects. The environment pointer, environ, references the data structure specified in a previous call to HOOK\_setenv.

**See Also**                      HOOK\_setenv  
                                  TSK\_getenv

**HOOK\_setenv***Set environment pointer for a given HOOK and TSK combination***C Interface****Syntax**`HOOK_setenv(task, id, environ);`**Parameters**

`TSK_Handle task;`      */\* task object handle \*/*  
`HOOK_Id id;`      */\* HOOK instance id \*/*  
`Ptr environ;`      */\* environment pointer \*/*

**Return Value**`Void`**Reentrant**`yes`**Description**

HOOK\_setenv sets the environment pointer associated with the specified HOOK and TSK objects to environ. The environment pointer, environ, should reference an data structure to be used by the hook functions for a task or tasks.

Each HOOK object may have a separate environment pointer for each task. A HOOK object may also point to the same data structure for all tasks, depending on its data sharing needs.

The HOOK\_getenv function can be used to get the environ pointer for a particular HOOK and TSK object combination.

**See Also**

HOOK\_getenv  
TSK\_setenv

## 2.11 HST Module

**Important Note:** This module is being deprecated and will no longer be supported in the next major release of DSP/BIOS.

The HST module is the host channel manager.

### Functions

□ HST\_getpipe. Get corresponding pipe object

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the HST Manager Properties and HST Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default (Enum Options)
OBJMEMSEG	Reference	prog.get("IDRAM")
HOSTLINKTYPE	EnumString	"RTDX" ("NONE")

#### Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
mode	EnumString	"output" ("input")
bufSeg	Reference	prog.get("IDRAM")
bufAlign	Int16	4
frameSize	Int16	128
numFrames	Int16	2
statistics	Bool	false
availableForDHL	Bool	false
notifyFxn	Extern	prog.extern("FXN_F_nop")
arg0	Arg	3

### 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.



**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 in the HST Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script. To create an HST object in a configuration script, use the following syntax:

```
var myHst = bios.HST.create("myHst");
```

The Tconf examples that follow assume the object has been created as shown.

- ❑ **comment.** A comment to identify this HST object.

Tconf Name: comment

Type: String

Example: `myHst.comment = "my HST";`

- ❑ **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.

Tconf Name: mode

Type: EnumString

Options: "output", "input"

Example: `myHst.mode = "output";`

- ❑ **bufseg.** The memory segment from which the buffer is allocated; all frames are allocated from a single contiguous buffer (of size framesize x numframes).

Tconf Name: bufSeg

Type: Reference

Example: `myHst.bufSeg = prog.get("myMEM");`

- ❑ **bufalign.** The alignment (in words) of the buffer allocated within the specified memory segment.

Tconf Name: bufAlign

Type: Int16

Options: must be  $\geq 4$  and a power of 2

Example: `myHst.bufAlign = 4;`

- ❑ **framesize.** The length of each frame (in words)

Tconf Name: frameSize

Type: Int16

Example: `myHst.frameSize = 128;`

- ❑ **numframes.** The number of frames

Tconf Name: numFrames

Type: Int16

Example: `myHst.numFrames = 2;`



- ❑ **statistics.** Set this property to true 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.

Tconf Name: statistics

Type: Bool

Example: `myHst.statistics = false;`

- ❑ **Make this channel available for a new DHL device.** Set this property to true 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.

Tconf Name: availableForDHL

Type: Bool

Example: `myHst.availableForDHL = false;`

- ❑ **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.

Tconf Name: notifyFxn

Type: Extern

Example: `myHst.notifyFxn =  
          prog.extern("hstNotify");`

- ❑ **arg0, arg1.** Two 32-bit arguments passed to the notify function. They can be either unsigned 32-bit constants or symbolic labels.

Tconf Name: arg0

Type: Arg

Tconf Name: arg1

Type: Arg

Example: `myHst.arg0 = 3;`

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

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS.

---

**C Interface****Syntax**`pipe = HST_getpipe(hst);`**Parameters**`HST_Handle hst`        */\* host object handle \*/***Return Value**`PIP_Handle pip`        */\* pipe object handle\*/***Reentrant**`yes`**Description**

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`

## 2.12 HWI Module

The HWI module is the hardware interrupt manager.

### Functions

- ❑ `HWI_applyWugenMasks`. Set WUGEN interrupt mask registers.
- ❑ `HWI_disable`. Disable hardware interrupts
- ❑ `HWI_disableWugen`. Disable an interrupt in WUGEN registers.
- ❑ `HWI_dispatchPlug`. Plug the HWI dispatcher
- ❑ `HWI_enable`. Enable hardware interrupts
- ❑ `HWI_enableWugen`. Enable an interrupt in WUGEN registers.
- ❑ `HWI_enter`. Hardware ISR prolog
- ❑ `HWI_eventMap`. Assign interrupt source number to an HWI object.
- ❑ `HWI_exit`. Hardware ISR epilog
- ❑ `HWI_getWugenMasks`. Get WUGEN interrupt mask registers.
- ❑ `HWI_ierToWugenMasks`. Compute WUGEN masks to match IER register.
- ❑ `HWI_isHWI`. Check current thread calling context.
- ❑ `HWI_restore`. Restore hardware interrupt state

### Constants, Types, and Structures

```
typedef struct HWI_Attrs {
    Uns    intrMask; /* IER bitmask, 1="self" (default) */
    Uns    ccMask    /* CSR CC bitmask, 1="leave alone" */
    Arg    arg;      /* fxn arg (default = 0) */
} HWI_Attrs;

HWI_Attrs HWI_ATTRS = {
    1,          /* interrupt mask (1 => self) */
    1,          /* CSR bit mask (1 => leave alone) */
    0           /* argument to ISR */
};
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the HWI Manager Properties and HWI Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters.

Name	Type	Default (Enum Options)
RESETVECTOR	Bool	false
EXTPIN4POLARITY	EnumString	"low-to-high" ("high-to-low")

Name	Type	Default (Enum Options)
EXTPIN5POLARITY	EnumString	"low-to-high" ("high-to-low")
EXTPIN6POLARITY	EnumString	"low-to-high" ("high-to-low")
EXTPIN7POLARITY	EnumString	"low-to-high" ("high-to-low")
ENABLEEXC	Bool	true (C64x+ only)

### Instance Configuration Parameters

HWI instances are provided as a default part of the configuration and cannot be created. In the items that follow, HWI\_INT\* may be any provided instance. Default values for many HWI properties are different for each instance.

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
interruptSource	EnumString	"Reset" (Non_Maskable", "Reserved", "Timer 0", "Timer 1", "Host_Port_Host_to_DSP", "EMIF_SDRAM_Timer", "PCI_WAKEUP", "AUX_DMA_HALT", "External_Pin_4", "External_Pin_5", "External_Pin_6", "External_Pin_7", "DMA_Channel_0", "DMA_Channel_1", "DMA_Channel_2", "DMA_Channel_3", "MCSP_0_Transmit", "MCSP_0_Receive", "MCSP_1_Transmit", "MCSP_2_Receive", "MCSP_2_Transmit", "MCSP_2_Receive") (Not used for C64x+)
interruptSelectNumber	Int	(varies by specific target)
fxn	Extern	prog.extern("HWI_unused","asm")
monitor	EnumString	"Nothing" ("Data Value", "Stack Pointer", "Top of SW Stack", "A0" ... "A15", "B0" ... "B15")
addr	Arg	0x00000000
dataType	EnumString	"signed" ("unsigned")
operation	EnumString	"STS_add(*addr)" ("STS_delta(*addr)", "STS_add(-*addr)", "STS_delta(-*addr)", "STS_add( *addr )", "STS_delta( *addr )")
useDispatcher	Bool	false
arg	Arg	0
interruptMask	EnumString	"self" ("all", "none", "bitmask")
interruptBitMask	Numeric	0x0010 *
cacheControl	Bool	true (Not used for C64x+)
progCacheMask	EnumString	"mapped" ("cache enable", "cache freeze", "cache bypass") (Not used for C64x+)
dataCacheMask	EnumString	"mapped" ("cache enable", "cache freeze", "cache bypass") (Not used for C64x+)

\* Depends on interrupt ID

## Description

The HWI module manages hardware interrupts. Using Tconf, 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 C62\_plug or C64\_plug functions.

DSP/BIOS supports the C64x+ interrupt selector through the HWI Module. You can route one of the 128 system events to a specific HWI object by specifying the event number as the "interrupt selection number" in the HWI Object Properties. This one-to-one mapping supports up to 12 maskable interrupts. The C64x+ event combiner is supported by the ECM Module. If the 16 HWI objects are sufficient for the number of HWI functions your application needs to run, you need not enable the ECM module. You do not need to know whether your C function will be run by the HWI module or ECM module when you write the function. (The ECM module uses the HWI dispatcher, and so its functions cannot be completely written in assembly.)

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.

---

### Note: RTS Functions Callable from TSK Threads Only

Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK\_pend and LCK\_post. As a result, RTS functions that call LCK\_pend or LCK\_post *must not be called in the context of a SWI or HWI thread*. For a list of RTS functions that should not be called from a SWI or an HWI function, see "LCK\_pend" on page 2-210.

---

The C++ "new" operator calls malloc, which in turn calls LCK\_pend. As a result, the "new" operator cannot be used in the context of a SWI or HWI thread.

### HWI Dispatcher vs. HWI\_enter/exit

The HWI dispatcher is the preferred method for handling an interrupt. When enabled, the HWI objects that run functions for the CLK and RTDX modules use the dispatcher.

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 a 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 HWI\_exit epilog, and the HWI function can be completely written in C. It would, in fact, cause a system crash for 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 the HWI dispatcher when it runs CLK\_F\_isr. Additionally, CLK functions should **not** use the *interrupt* keyword or the INTERRUPT pragma in C functions.

---

**Notes**

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.

- ❑ Do not call SWI\_disable or SWI\_enable within an HWI function.
- ❑ Do not call HWI\_enter, HWI\_exit, or any other DSP/BIOS functions from a non-maskable interrupt (NMI) service routine. In addition, the HWI dispatcher cannot be used with the NMI service routine.

In general, due to details of the 'C6000 architecture, NMI disrupts the code it interrupts to the point that it cannot be returned to. Therefore, NMI should not be used to respond to run-time events. NMI should be used only for exceptional processing that does not return to the code it interrupted.

- ❑ 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

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.

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 SWI scheduling by calling HWI\_enter and HWI\_exit.

## Registers and Stack

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.

The register mask argument to HWI\_enter and HWI\_exit allows you to save and restore registers used within the function. Other arguments, for example, allow the HWI to control the settings of the IEMASK and the cache control field.

---

### 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. You can use the IEMASK parameter for the HWI\_enter API 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 properties can be set for the HWI module in the HWI Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ☐ **Generate RESET vector at address 0.** Check this box in order to place an additional reset vector at address 0. You need to enable this property only if you generated your vector table somewhere other than address 0 but want the reset vector to be at address 0. This option is available only if address 0 exists in the memory configuration and the .hwi\_vec section is not placed in a memory segment containing address 0.

Tconf Name: RESETVECTOR

Type: Bool

Example: `bios.HWI.RESETVECTOR = false;`

- ☐ **External Interrupt Pin 4-7 Polarity.** Choose whether the device connected to this pin causes an interrupt when a high-to-low transition occurs, or when a low-to-high transition occurs.

Tconf Name: EXTPIN4POLARITY

Type: EnumString

Tconf Name: EXTPIN5POLARITY

Type: EnumString

Tconf Name: EXTPIN6POLARITY

Type: EnumString

Tconf Name: EXTPIN7POLARITY

Type: EnumString

Options: "low-to-high", "high-to-low"

Example: `bios.HWI.EXTPIN4POLARITY = "low-to-high";`

- ☐ **Enable EXC module exception processing.** C64x+ only. Leave this property set to true if you plan to use the EXC or MPC Module. By default, the EXC module is enabled.

Tconf Name: ENABLEEXC

Type: Bool

Example: `bios.HWI.ENABLEEXC = true;`

## HWI Object Properties

The following properties can be set for an HWI object in the HWI Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script. The HWI objects for the platform are provided in the default configuration and cannot be created.

- ☐ **comment.** A comment is provided to identify each HWI object.

Tconf Name: comment

Type: String

Example: `bios.HWI_INT4.comment = "myISR";`



- ❑ **interrupt source.** Select the pin, DMA channel, timer, or other source of the interrupt. Only the most common sources are listed. If your source is not listed here as an option, use the interrupt selection number property instead. (Not used for C64x+ devices.)

Tconf Name: interruptSource

Type: EnumString

Options: "Reset", "Non\_Maskable", "Reserved", "Timer 0",  
"Timer 1", "Host\_Port\_Host\_to\_DSP",  
"EMIF\_SDRAM\_Timer", "PCI\_WAKEUP",  
"AUX\_DMA\_HALT", "External\_Pin\_4",  
"External\_Pin\_5", "External\_Pin\_6", "External\_Pin\_7",  
"DMA\_Channel\_0", "DMA\_Channel\_1",  
"DMA\_Channel\_2", "DMA\_Channel\_3",  
"MCSP\_0\_Transmit", "MCSP\_0\_Receive",  
"MCSP\_1\_Transmit", "MCSP\_2\_Receive",  
"MCSP\_2\_Transmit", "MCSP\_2\_Receive"

```
Example: bios.HWI_INT4.interruptSource =
         "External Pin 4";
```

- ❑ **interrupt selection number.** The source number associated with an interrupt. This property overrides the interrupt source selection, and should be used if your interrupt source is not listed as an option for the previous property. This value is used to program the interrupt multiplexer registers or the interrupt selector. You can use the HWI\_eventMap API to change this property at run-time. To use this property with the C64x+ event combiner, see Section 2.7, *ECM Module*.

Tconf Name: interruptSelectionNumber

Type: Int

Example: bios.HWI INT4.interruptSelectionNumber=4;

- ❑ **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 a SWI or a semaphore. HWI functions can post SWIs, 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.

Tconf Name: fxn

Type: Extern

```
Example: bios.HWI_INT4.fxn = prog.extern("myHWI",
                                           "asm");
```

- ❑ **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 HWI function. 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.

Options: "Nothing", "Data Value", "Stack Pointer", "Top of SW Stack", "A0" ... "A15", "B0" ... "B15"

Example: `bios.HWI_INT4.monitor = "Nothing";`

- ❑ **addr.** If the monitor property above is set to Data Address, this property 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.

Tconf Name: `addr` Type: Arg

Example: `bios.HWI_INT4.addr = 0x00000000;`

- ❑ **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.

Tconf Name: `dataType` Type: EnumString

Options: "signed", "unsigned"

Example: `bios.HWI_INT4.dataType = "signed";`

- ❑ **operation.** The operation to be performed on the value monitored. You can choose one of several STS operations.

Tconf Name: `operation` Type: EnumString

Options: "STS\_add(\*addr)", "STS\_delta(\*addr)", "STS\_add(-\*addr)", "STS\_delta(-\*addr)", "STS\_add(|\*addr|)", "STS\_delta(|\*addr|)"

Example: `bios.HWI_INT4.operation = "STS_add(*addr)";`

- ❑ **Use Dispatcher.** A check box that controls whether the HWI dispatcher is used. The HWI dispatcher cannot be used for the non-maskable interrupt (NMI) service routine.

Tconf Name: `useDispatcher` Type: Bool

Example: `bios.HWI_INT4.useDispatcher = false;`

- ❑ **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.

Tconf Name: arg

Type: Arg

Example: `bios.HWI_INT4.arg = 3;`

- ❑ **Interrupt Mask.** Specifies which interrupts the dispatcher should disable before calling the function. This property is available only when using the HWI dispatcher.

- The "self" option causes the dispatcher to disable only the current interrupt.

- The "all" option disables all interrupts.

- The "none" option disables no interrupts.

- The "bitmask" option causes the interruptBitMask property to be used to specify which interrupts to disable.

Tconf Name: interruptMask

Type: EnumString

Options: "self", "all", "none", "bitmask"

Example: `bios.HWI_INT4.interruptMask = "self";`

- ❑ **Interrupt Bit Mask.** An integer property that is writable when the interrupt mask is set to "bitmask". This should be a hexadecimal integer bitmask specifying the interrupts to disable.

Tconf Name: interruptBitMask

Type: Numeric

Example: `bios.HWI_INT4.interruptBitMask = 0x0010;`

Options: "self", "all", "none", "bitmask"

- ❑ **Don't modify cache control.** (Not used for C64x+) A check box that chooses between not modifying the cache at all or enabling the individual drop-down menus for program and data cache control masks. This property is available only when using the HWI dispatcher. This property and the two that follow are not used for C64x+ because the HWI dispatcher does not perform cache control for C64x+.

Tconf Name: cacheControl

Type: Bool

Example: `bios.HWI_INT4.cacheControl = true;`

- ❑ **Program Cache Control Mask.** (Not used for C64x+) A drop-down menu that becomes writable when the “don’t modify cache control” property is set to false. The choices are the same choices available from the GBL properties.

Tconf Name: progCacheMask Type: EnumString

Options: "mapped", "cache enable", "cache freeze", "cache bypass"

Example: `bios.HWI_INT4.progCacheMask = "mapped";`

- ❑ **Data Cache Control Mask.** (Not used for C64x+) A drop-down menu that becomes writable when the “don’t modify cache control” property is set to false. The choices are the same choices available from the “program cache control mask” menu.

Tconf Name: dataCacheMask                      Type: EnumString

Options: "mapped", "cache enable", "cache freeze", "cache bypass"

**Example:** `bios.HWI_INT4.dataCacheMask = "mapped";`

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 that uses the dispatcher. As a result, you can modify the dispatcher's parameters for the CLK HWI, such as the cache setting or the interrupt mask. However, you cannot disable use of the dispatcher for the CLK HWI.

Table 2-4 lists these precreated objects and their default interrupt sources. The HWI object names are the same as the interrupt names.

*Table 2-4. HWI interrupts for the TMS320C6000*

Name	Default Interrupt Source
HWI_RESET	Reset
HWI_NMI	NMI
HWI_RESERVED0	
HWI_RESERVED1	
HWI_INT4	INT4
HWI_INT5	INT5
HWI_INT6	INT6

---

<b>Name</b>	<b>Default Interrupt Source</b>
HWI_INT7	INT7
HWI_INT8	INT8
HWI_INT9	INT9
HWI_INT10	INT10
HWI_INT11	INT11
HWI_INT12	INT12
HWI_INT13	INT13
HWI_INT14	INT14
HWI_INT15	INT15

---

**HWI\_applyWugenMasks***Apply specified masks to WUGEN interrupt mask registers***C Interface****Syntax**                      HWI\_applyWugenMasks(mask[]);**Parameters**                Uint32            mask[];    /\* array of masks to apply to WUGEN registers \*/**Return Value**              Void**Reentrant**                    yes**Description**                This function is available only for OMAP 2430/3430 devices.

HWI\_applyWugenMasks applies the specified masks to the WUGEN interrupt mask registers. The WUGEN registers are the Wakeup Generator registers.

If a bit in a mask is enabled, the corresponding interrupt in the WUGEN will be blocked.

The mask[] array should contain the following masks in four integers:

- ❑ First mask consists of IRQ 0-31
- ❑ Second mask consists of IRQ 32-47
- ❑ Third mask consists of DMA requests
- ❑ Fourth mask consists of hpi access wake-up

For details about WUGEN registers, see literature item number SWPU090 (for OMAP 2430) and number SWPU100 (for OMAP 3430).

**Constraints and  
Calling Context**

- ❑ This function should be called with interrupts disabled. This ensures that when interrupts are re-enabled, the pending interrupt with the highest priority is executed first.

**See Also**

HWI\_getWugenMasks  
HWI\_disableWugen  
HWI\_enableWugen  
HWI\_ierToWugenMasks

**HWI\_disable***Disable hardware interrupts***C Interface****Syntax**                    `oldCSR = HWI_disable();`**Parameters**                `Void`**Return Value**              `Uns oldCSR;`**Reentrant**                   `yes`**Description**

HWI\_disable disables hardware interrupts by clearing the GIE bit in the Control Status Register (CSR). 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.

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.

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

HWI\_disable may be called from main(). However, since HWI interrupts are already disabled in main(), such a call has no effect.

**Example**

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

**See Also**

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

**HWI\_disableWugen**

*Disable an event in the WUGEN interrupt mask registers*

**C Interface**

<b>Syntax</b>	HWI_disableWugen(eventid);
<b>Parameters</b>	Int            eventid;    /* event number associated with the interrupt */
<b>Return Value</b>	Void
<b>Reentrant</b>	yes
<b>Description</b>	<p>This function is available only for OMAP 2430/3430 devices.</p> <p>HWI_disableWugen disables the interrupt source you specify. It sets the appropriate bit in the WUGEN registers.</p> <p>Use the eventid parameter to specify an event using the event number associated with an interrupt.</p>
<b>Constraints and Calling Context</b>	None
<b>See Also</b>	HWI_enableWugen HWI_applyWugenMasks HWI_getWugenMasks HWI_ierToWugenMasks



**HWI\_dispatchPlug***Plug the HWI dispatcher***C Interface****Syntax**

HWI\_dispatchPlug(vecid, fxn, dmachan, attrs);

**Parameters**

Int            vecid;        /\* interrupt id \*/  
 Fxn           fxn;         /\* pointer to HWI function \*/  
 Int           dmachan; /\* DMA channel to use for performing plug \*/  
 HWI\_Attrs    \*attrs       /\*pointer to HWI dispatcher attributes \*/

**Return Value**

Void

**Reentrant**

yes

**Description**

HWI\_dispatchPlug fills the HWI dispatcher table with the function specified by the fxn parameter and the attributes specified by the attrs parameter.

If the specified interrupt (vecid) was not configured to be dispatched (via Tconf or Gconf configuration), then HWI\_dispatchPlug writes an Interrupt Service Fetch Packet (ISFP) into the Interrupt Service Table (IST), at the address corresponding to vecid. The op-codes written in the ISFP create a branch to the HWI dispatcher. If the interrupt was previously configured to be dispatched, then the HWI dispatcher table is still updated using the fxn and attrs parameters, but a new ISFP is not written to the IST.

The dmachan is needed only for 'C6x0x devices if the IST is located in internal program RAM. Since the 'C6x0x CPU cannot write to internal program RAM, it needs to use DMA to write to IPRAM. This is not the case for 'C6x1x and 'C64x devices.

For 'C6x0x devices, if the IST is stored in external RAM, a DMA (Direct Memory Access) channel is not necessary and the dmachan parameter can be set to -1 to cause a CPU copy instead. A DMA channel can still be used to plug a vector in external RAM. A DMA channel must be used to plug a vector in internal program RAM.

For 'C6x11 and 'C64x devices, you may set the dmachan parameter to -1 to specify a CPU copy, regardless of where the IST is stored. Alternately, you may specify the DMA channel.

For 'C64x+ devices, the dmachan is ignored. However, there is a case where DMA is automatically used by HWI\_dispatchPlug on 'C64x+ devices. If the vector table location is L1P SRAM, then IDMA1 is used for the vector copy. In this case, HWI\_dispatchPlug waits for any activity to finish on IDMA1 before using it. It then waits for the vector copy DMA

activity to complete before returning. Since the stack is used for the source location of the DMA copy, HWI\_dispatchPlug must be called while a stack from internal memory (L1 or L2) is active (and only when the vector table is in L1P SRAM).

If you use the dmachan parameter to specify a DMA channel, HWI\_dispatchPlug assumes that the DMA channel is available for use, and stops the DMA channel before programming it. If the DMA channel is shared with other code, use a semaphore or other DSP/BIOS signaling method to provide mutual exclusion before calling HWI\_dispatchPlug, C62\_plug, or C64\_plug.

HWI\_dispatchPlug does not enable the interrupt. Use C62\_enableIER or C64\_enableIER to enable specific interrupts.

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.

```
typedef struct HWI_Attrs {
    Uns    intrMask; /* IER bitmask, 1="self" (default) */
    Uns    ccMask    /* CSR CC bitmask, 1="leave alone" */
    Arg    arg;      /* fxn arg (default = 0) */
} HWI_Attrs;
```

The intrMask element is a bitmask that specifies which interrupts to mask off while executing the HWI. Bit positions correspond to those of the IER. A value of 1 indicates an interrupt is being plugged. The default value is 1.

For most C6000 platforms, the ccMask element is a bitfield that corresponds to the cache control bitfield in the CSR. A value of 1 indicates that the HWI dispatcher should not modify the cache control settings at all. The default value is 1.

For C64x+ devices, the ccMask element is ignored, since no cache handling occurs within the HWI dispatcher.

The default values are defined as follows:

```
HWI_Attrs HWI_ATTRS = {
    1,    /* interrupt mask (1 => self) */
    1,    /* CSR bit mask (1 => leave alone) */
    0     /* argument to ISR */
};
```

The arg element is a generic argument that is passed to the plugged function as its only parameter. The default value is 0.

## Constraints and Calling Context

- ❑ vecid must be a valid interrupt ID in the range of 0-15.

- ❑ dmachan must be 0, 1, 2, or 3 if the IST is in internal program memory and the device is a 'C6x0x.

**See Also**

HWI\_enable  
HWI\_restore  
C62\_plug  
C64\_plug  
HWI\_eventMap  
SWI\_disable  
SWI\_enable

**HWI\_enable***Enable interrupts***C Interface****Syntax** HWI\_enable();**Parameters** Void**Return Value** Void**Reentrant** yes**Description** HWI\_enable enables hardware interrupts by setting the GIE bit in the Control Status Register (CSR).

Hardware interrupts are enabled unless a call to HWI\_disable disables them. DSP/BIOS enables hardware interrupts after the program's main() function runs. Your main() function can enable individual interrupt mask bits, but it should not call HWI\_enable to globally enable interrupts.

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\_enableWugen***Enable an event in the WUGEN interrupt mask registers***C Interface****Syntax**                      HWI\_enableWugen(eventid);**Parameters**                Int                eventid;    /\* event number associated with the interrupt \*/**Return Value**                Void**Reentrant**                    yes**Description**                This function is available only for OMAP 2430/3430 devices.

HWI\_enableWugen enables the interrupt source you specify. It clears the appropriate bit in the WUGEN registers.

Use the eventid parameter to specify an event using the event number associated with an interrupt.

**Constraints and  
Calling Context**                None**See Also**                      HWI\_disableWugen  
                                 HWI\_applyWugenMasks  
                                 HWI\_getWugenMasks  
                                 HWI\_ierToWugenMasks

HWI\_enter

Hardware ISR prolog

C Interface

Syntax	none
Parameters	none
Return Value	none

Assembly Interface

Syntax	HWI_enter AMASK, BMASK, CMASK, IEMASK, CCMASK
Preconditions	interrupts are globally disabled (that is, GIE == 0)
Postconditions	amr = 0 GIE = 1 dp (b14) = .bss

**Modifies** a0, a1, a2, a3, amr, b0, b1, b2, b3, b14, b15, csr, ier

**Reentrant** yes

**Description** HWI\_enter is an API (assembly macro) used to save the appropriate context for a DSP/BIOS hardware interrupt (HWI).

The arguments to HWI\_enter are bitmasks that define the set of registers to be saved and bitmasks that define which interrupts are to be masked during the execution of the HWI.

HWI\_enter is used by HWIs that are user-dispatched, as opposed to HWIs that are handled by the HWI dispatcher. HWI\_enter must not be issued by HWIs 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 HWI before any DSP/BIOS API calls that could trigger other DSP/BIOS objects, such as posting a 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 HWIs that call C functions.

Common masks are defined in the device-specific assembly macro file c6x.h62. This file defines C6X\_ATEMPS, C6X\_BTEMPS, and C6X\_CTEMPS. These masks specify the C temporary registers and should be used when saving the context for an HWI that is written in C.

The c62.h62 and c64.h64 files define deprecated C62\_ and C64\_ masks for backward compatibility. Code that uses the old C62\_ABTEMPs mask will compile correctly, but will generate a warning.

The input parameter CCMASK specifies the program cache control (PCC) and data cache control (DCC) codes you need to use in the context of the HWI. Some typical values for this mask are defined in c6x.h62. The PCC code and DCC code can be ORed together (for example, C6X\_PCC\_ENABLE | C6X\_PCC\_DISABLE) to generate CCMASK.

The following parameters and constants are available for HWI\_enter. These match the parameters used for HWI\_exit, except that IEMASK corresponds to IERRESTOREMASK.

- ❑ **AMASK, BMASK.** Register mask specifying A, B registers to save
  - **C6X\_ATEMPS, C6X\_BTEMPs.** Masks to use if calling a C function from within an HWI; defined in c6x.h62.
  - **C6X\_A0 to C6X\_A15, C6X\_B0 to C6X\_B15.** For 'C62x and 'C67x platforms. Individual register constants; can be ORed together for more precise control than using C6X\_ATEMPS and C6X\_BTEMPs.
  - **C6X\_A0 to C6X\_A31, C6X\_B0 to C6X\_B31.** For 'C64x, 'C64x+, and 'C67+ platforms. Individual register constants; can be ORed together for more precise control than using C6X\_ATEMPS and C6X\_BTEMPs.
- ❑ **CMASK.** Register mask specifying control registers to save
  - **C6X\_CTEMPs.** Mask to use if calling a C function from within an HWI. Defined in c6x.h62.
  - **C6X\_AMR, C6X\_CSR, C6X\_IER, C6X\_IST, C6X\_IRP, C6X\_NRP.** Individual register constants; can be ORed together for more precise control than using C6X\_CTEMPs.
- ❑ **IEMASK.** Bit mask specifying IER bits to disable. Any bit mask can be specified, with bits having a one-to-one correspondence with the assigned values in the IER. The following convenience macros can be ORed together to specify the mask of interrupts to disable
  - **C6X\_NMIE**
  - **C6X\_IE4 to C6X\_IE15**

- ❑ **CCMASK.** Bit mask specifying cache control bits in the CSR. The following macros directly correspond to the possible modes of the program cache specified in the CSR. (Although the HWI dispatcher does not support cache control for C64x+ devices, the CCMASK is used for C64x+ devices with HWI\_enter.)

- **C6X\_PCC\_DISABLE**

- **C6X\_PCC\_ENABLE**

- **C6X\_PCC\_FREEZE**

- **C6X\_PCC\_BYPASS**

Note that if HWI\_enter modifies CSR bits, those changes are lost when interrupt processing is complete. HWI\_exit restores the CSR to its value when interrupt processing began no matter what the value of CCMASK.

## Constraints and Calling Context

- ❑ This API should not be used in 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 HWI that uses any DSP/BIOS API calls that might post or affect a SWI 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.
- ❑

## Example

CLK\_isr:

```
HWI_enter C6X_ATEMPS, C6X_BTEMPS, C6X_CTEMPS, 0XF0, \
C6X_PCC_ENABLE|C6X_PCC_DISABLE
PRD_tick
HWI_exit C6X_ATEMPS, C6X_BTEMPS, C6X_CTEMPS, 0XF0, \
C6X_PCC_ENABLE|C6X_PCC_DISABLE
```

## See Also

HWI\_exit



**HWI\_eventMap***Assign interrupt selection number to HWI object***C Interface****Syntax**

HWI\_eventMap(vectID, eventID);

**Parameters**

Int            vectID;    /\* number of HWI object (indexed from 0) \*/  
 Int            eventID; /\* event or interrupt selection number \*/

**Return Value**

Void

**Reentrant**

yes

**Description**

This function is available only for C6000 devices.

HWI\_eventMap associates an interrupt source selection number (or an eventID) to a specific HWI vector ID. This causes the specified interrupt event to invoke the specified HWI object.

This API allows you to remap an interrupt source to a particular HWI object at run-time. To statically configure the same thing, use the interruptSelectionNumber or interruptSource HWI instance property.

The vectID parameter is the Hardware Vector number. For example, HWI\_INT4 has a vectID of 4, and HWI\_INT5 has a vectID of 5.

The eventID parameter should match the event ID. For C64x+ platforms, these events are listed in the "System Event Mapping" table in the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871). Device-specific data manuals contain additional information about event numbers. For other C6000 devices, you can select the interrupt source you want in the DSP/BIOS Configuration Tool to see its corresponding interrupt selection number.

This API is protected by calls to HWI\_disable/HWI\_restore, so it can be called from any thread, including an HWI thread.

**Constraints and Calling Context**

None

**Example**

```
// maps interrupt selection number 1 to HWI object 14
HWI_eventMap(14, 1);
```

**See Also**

HWI\_dispatchPlug  
 C62\_plug  
 C64\_plug

**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 AMASK BMASK CMASK IERRESTOREMASK CCMASK
<b>Preconditions</b>	b14 = pointer to the start of .bss amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, amr, b0, b1, b2, b3, b14, b15, csr, ier, irp

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

<b>Description</b>	HWI_exit is an API (assembly macro) which is used to restore the context that existed before a DSP/BIOS hardware interrupt (HWI) was invoked.
--------------------	---

HWI\_exit is used by HWIs that are user-dispatched, as opposed to HWIs that are handled by the HWI dispatcher. HWI\_exit must not be issued by HWIs 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 HWI that uses DSP/BIOS API calls which could trigger other DSP/BIOS objects, such as posting a SWI or semaphore.

HWI\_exit restores the registers specified by AMASK, BMASK, and CMASK. These masks are 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 HWIs that call C functions only for HWIs that are not dispatched by the HWI dispatcher.

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

Of the interrupts in IERRESTOREMASK, HWI\_exit only restores those enabled upon entering the HWI. HWI\_exit does not affect the status of interrupt bits that are not in IERRESTOREMASK.

- ❑ If upon exiting an HWI you do not wish to restore an interrupt that was disabled with HWI\_enter, do not set that interrupt bit in the IERRESTOREMASK in HWI\_exit.
- ❑ If upon exiting an HWI you wish to enable an interrupt that was disabled upon entering the HWI, set the corresponding bit in IER register. (Including a bit in IER in the IERRESTOREMASK of HWI\_exit does not enable the interrupt if it was disabled when the HWI was entered.)

For a list of parameters and constants available for use with HWI\_exit, see the description of HWI\_enter. In addition, see the c6x.h62 file.

To be symmetrical, even though CCMASK has no effect on HWI\_exit, you should use the same CCMASK that is used in HWI\_enter for HWI\_exit. HWI\_exit restores the CSR to its value when interrupt processing began no matter what the value of CCMASK.

## 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 function uses the HWI dispatcher.
- ❑ If the HWI dispatcher is not used, this API must be the last operation in an HWI that uses any DSP/BIOS API calls that might post or affect a SWI 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.
- ❑ The AMASK, BMASK, and CMASK 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 a SWI, TSK, or IDL function.
- ❑ This API cannot be called from a CLK function.

## Example

CLK\_isr:

```
HWI_enter C6X_ATEMPS, C6X_BTEMPS, C6X_CTEMPS, 0XF0, \
C6X_PCC_ENABLE|C6X_PCC_DISABLE
PRD_tick
HWI_exit C6X_ATEMPS, C6X_BTEMPS, C6X_CTEMPS, 0XF0, \
C6X_PCC_ENABLE|C6X_PCC_DISABLE
```

## See Also

HWI\_enter

**HWI\_getWugenMasks**

*Get masks from WUGEN interrupt mask registers*

**C Interface**

<b>Syntax</b>	HWI_getWugenMasks(mask[]);
<b>Parameters</b>	Uint32        mask[];    /* array of WUGEN masks */
<b>Return Value</b>	Void

**Reentrant**                yes

**Description**            This function is available only for OMAP 2430/3430 devices.

HWI\_getWugenMasks gets the WUGEN interrupt mask registers.

If a bit in a mask is enabled, the corresponding interrupt in the WUGEN is blocked.

After this function returns, the mask[] array will contain the following masks in four integers:

- ❑ First mask consists of IRQ 0-31
- ❑ Second mask consists of IRQ 32-47
- ❑ Third mask consists of DMA requests
- ❑ Fourth mask consists of hpi access wake-up

**Constraints and  
Calling Context**                None

**See Also**                    HWI\_disableWugen  
                                HWI\_enableWugen  
                                HWI\_applyWugenMasks  
                                HWI\_ierToWugenMasks

**HWI\_ierToWugenMasks***Compute WUGEN masks from IER register***C Interface****Syntax** HWI\_ierToWugenMasks(mask[]);**Parameters** Uint32 mask[]; /\* array of WUGEN masks \*/**Return Value** Void**Reentrant** yes**Description** This function is available only for OMAP 2430/3430 devices.

HWI\_ierToWugenMasks computes the WUGEN masks needed to allow the interrupts set in the IER register to propagate through the WUGEN.

This function does not enable external DMA requests that are routed directly to the EDMA but are not set in the IER registers. In fact, these will be blocked in the masks returned by this function. To enable such DMA requests as a wakeup event, you must set the corresponding bits in the WUGEN masks returned by this function, before using the masks in a call to HWI\_applyWugenMasks().

The mask[] array contains the following masks in four integers:

- ❑ First mask consists of IRQ 0-31
- ❑ Second mask consists of IRQ 32-47
- ❑ Third mask consists of DMA requests
- ❑ Fourth mask consists of hpi access wake-up

This function does not set any WUGEN registers, it simply computes the mask[] values. To apply the computed WUGEN masks, call HWI\_applyWugenMasks with the mask[] array values returned by HWI\_ierToWugenMasks.

**Constraints and Calling Context** None

**See Also**

HWI\_disableWugen  
 HWI\_enableWugen  
 HWI\_applyWugenMasks  
 HWI\_getWugenMasks

**HWI\_isHWI***Check to see if called in the context of an HWI***C Interface****Syntax**

```
result = HWI_isHWI(Void);
```

**Parameters**

Void

**Return Value**

Bool            result;     /\* TRUE if in HWI context, FALSE otherwise \*/

**Reentrant**

yes

**Description**

This macro returns TRUE when it is called within the context of an HWI or CLK function. This macro returns FALSE in all other contexts.

In previous versions of DSP/BIOS, calling HWI\_isHWI() from main() resulted in TRUE. This is no longer the case; main() is identified as part of the TSK context.

**See Also**

SWI\_isSWI

TSK\_isTSK

**HWI\_restore***Restore global interrupt enable state***C Interface****Syntax**                      HWI\_restore(oldCSR);**Parameters**                  Uns                  oldCSR;**Returns**                      Void**Reentrant**                    yes**Description**

HWI\_restore sets the global interrupt enable (GIE) bit in the Control Status Register (CSR) using the least significant bit of the oldCSR parameter. If bit 0 is 0, the GIE bit is not modified. If bit 0 is 1, the GIE bit is set to 1, which enables interrupts.

When you call HWI\_disable, the previous contents of the 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.

HWI\_restore may be called from main(). However, since HWI\_enable cannot be called from main(), interrupts are always disabled in main(), and a call to HWI\_restore has no effect.

**Constraints and Calling Context**

- ❑ HWI\_restore must be called with interrupts disabled. The parameter passed to HWI\_restore must be the value returned by HWI\_disable.

**Example**

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

**See Also**

HWI\_enable  
HWI\_disable

## 2.13 IDL Module

The IDL module is the idle thread manager.

### Functions

- ❑ `IDL_run`. Make one pass through idle functions.

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the IDL Manager Properties and IDL Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")
AUTOCALCULATE	Bool	true
LOOPINSTCOUNT	Int32	1000

#### Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
fxn	Extern	prog.extern("FXN_F_nop")
calibration	Bool	true
order	Int16	0

### 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 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 2–132), 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 HWIs or higher-priority tasks) is required.

## IDL Manager Properties

The following global properties can be set for the IDL module in the IDL Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: OBJMEMSEG Type: Reference

Example: `bios.IDL.OBJMEMSEG = prog.get("myMEM");`

- ☐ **Auto calculate idle loop instruction count.** When this property is set to true, the program runs the IDL functions one or more times 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 the 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 setting the "Include in CPU load calibration" property for an IDL object to false.

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

Tconf Name: AUTOCALCULATE Type: Bool

Example: `bios.IDL.AUTOCALCULATE = true;`

- ❑ **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.

Tconf Name: LOOPINSTCOUNT

Type: Int32

Example: `bios.IDL.LOOPINSTCOUNT = 1000;`

## 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.

To create an IDL object in a configuration script, use the following syntax. The Tconf examples assume the object is created as shown here.

```
var myId1 = bios.IDL.create("myId1");
```

The following properties can be set for an IDL object:

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

Tconf Name: comment

Type: String

Example: `myIdl.comment = "IDL function";`

- ❑ **function.** The function to execute. If this function is written in C and you use the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. (The DSP/BIOS Configuration Tool generates assembly code, which must use leading underscores when referencing C functions or labels.) If you use Tconf, do not add an underscore before the function name; Tconf adds the underscore to call a C function from assembly internally.

Tconf Name: fxn

Type: Extern

Example: `myIdl.fxn = prog.extern("myIDL");`

- ❑ **Include in CPU load calibration.** You can remove an individual IDL function from the CPU load calculation by setting this property to false. The CPU load calibration is performed only if the "Auto calculate idle loop instruction count" property is true 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.

Tconf Name: calibration

Type: Bool

Example: `myIdl.calibration = true;`

- ❑ **order.** Set this property for all IDL objects so that the numbers match the sequence in which IDL functions should be executed.

Tconf Name: order

Type: Int16

Example: `myId1.order = 2;`

**IDL\_run***Make one pass through idle functions***C Interface****Syntax** IDL\_run();**Parameters** Void**Return Value** Void**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.

## 2.14 LCK Module

Functions

The LCK module is the resource lock manager.

- ❑ 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 */
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the LCK Manager Properties and LCK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

Module Configuration Parameter.

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")

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 DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment that contains the LCK objects.

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.LCK.OBJMEMSEG = prog.get("myMEM");`

## LCK Object Properties

To create a LCK object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myLck = bios.LCK.create("myLck");
```

The following property can be set for a LCK object in the LCK Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: comment

Type: String

Example: `myLck.comment = "LCK object";`

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

```
lock = LCK_create(attrs);
```

**Parameters**

```
LCK_Attrs  attrs;    /* pointer to lock attributes */
```

**Return Value**

```
LCK_Handle lock;     /* handle for new lock object */
```

**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 2–240.

**Constraints and  
Calling Context**

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

**See Also**

LCK\_delete  
LCK\_pend  
LCK\_post

**LCK\_delete***Delete a resource lock***C Interface**

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

**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 a 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 Tconf, SYS\_error is called.

**See Also**

LCK\_create  
LCK\_pend  
LCK\_post

LCK\_pend

Acquire ownership of a resource lock

C Interface

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

**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 a timeout occurs before it can acquire ownership. LCK\_pend returns FALSE if it is called from the context of a SWI or HWI, even if the timeout is zero.

**Note: RTS Functions Callable from TSK Threads Only**

Many run-time support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK\_pend and LCK\_post. As a result, RTS functions that call LCK\_pend or LCK\_post *must not be called in the context of a SWI or HWI thread.*

To determine whether a particular RTS function uses LCK\_pend or LCK\_post, refer to the source code for that function shipped with Code Composer Studio. The following table lists some RTS functions that call LCK\_pend and LCK\_post in certain versions of Code Composer Studio:

fprintf	printf	vfprintf	sprintf
vprintf	vsprintf	clock	strftime
minit	malloc	realloc	free
calloc	rand	srand	getenv



The C++ new operator calls malloc, which in turn calls LCK\_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

**Constraints and  
Calling Context**

- ❑ The lock must be a handle for a resource lock object created through a prior call to LCK\_create.
- ❑ LCK\_pend should not be called from a SWI or HWI thread.
- ❑ LCK\_pend should not be called from main().

**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

**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.
- ❑ LCK\_post should not be called from a SWI or HWI thread.
- ❑ LCK\_post should not be called from main().

**See Also**

LCK\_create  
LCK\_delete  
LCK\_pend

## 2.15 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\_event5. Append 5-argument unformatted message to log.
- ❑ LOG\_message. Write a user message event to the system log.
- ❑ LOG\_printf. Append formatted message to message log.
- ❑ LOG\_printf4. Append 4-argument formatted message to log.
- ❑ LOG\_reset. Reset the system log.

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the LOG Manager Properties and LOG Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")
TS	Bool	false

#### Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
bufSeg	Reference	prog.get("IDRAM")
bufLen	EnumInt	64 (0, 8, 16, 32, 64, ..., 32768)
logType	EnumString	"circular" ("fixed")
dataType	EnumString	"printf" ("raw data")
format	String	"0x%x, 0x%x, 0x%x"

## 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 2–519, 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.

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 in the LOG Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.LOG.OBJMEMSEG = prog.get("myMEM");`

- ☐ **timestamped LOGs.** If enabled, timestamps are added to LOG records created by LOG APIs. The timestamp uses the same counter used by CLK\_gettime. When timestamping is enabled, each LOG record is 8 words. When timestamping is disabled, each LOG record is 4 words. You must enable timestamping in order to use the

LOG\_event5 and LOG\_printf4 APIs. When you enable timestamping, the logs cannot be handled by the DSP/BIOS plug-ins in CCStudio. However, timestamped logs are supported by the System Analyzer Tool.

Tconf Name: TS

Type: Bool

Example: `bios.LOG.TS = false;`

## LOG Object Properties

To create a LOG object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myLog = bios.LOG.create("myLog");
```

The following properties can be set for a log object on the LOG Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this LOG object.

Tconf Name: comment

Type: String

Example: `myLog.comment = "trace LOG";`

- ❑ **bufseg.** The name of a memory segment to contain the log buffer.

Tconf Name: bufSeg

Type: Reference

Example: `myLog.bufSeg = prog.get("myMEM");`

- ❑ **buflen.** The length of the log buffer (in words).

Tconf Name: bufLen

Type: EnumInt

Options: 0, 8, 16, 32, 64, ..., 32768

Example: `myLog.bufLen = 64;`

- ❑ **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.

Tconf Name: logType

Type: EnumString

Options: "circular", "fixed"

Example: `myLog.logType = "circular";`

- ❑ **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.

Tconf Name: dataType Type: EnumString

Options: "printf", "raw data"

Example: myLog.dataType = "printf";

- ❑ **format.** If you choose raw data as the datatype, type a printf-style format string for this property. 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, %u, %x, %o, %s, %r, and %p conversion characters; it cannot use other types of conversion characters. See LOG\_printf, page 2–224, and LOG\_event, page 2–220, for information about the structure of a log record.

Tconf Name: format Type: String

Example: myLog.format = "0x%x, 0x%x, 0x%x";

**LOG\_disable***Disable a message log***C Interface**

<b>Syntax</b>	LOG_disable(log);
<b>Parameters</b>	LOG_Handle log;      /* log object handle */
<b>Return Value</b>	Void
<b>Reentrant</b>	no
<b>Description</b>	LOG_disable disables the logging mechanism and prevents the log buffer from being modified.
<b>Example</b>	<pre>LOG_disable(&amp;trace);</pre>
<b>See Also</b>	LOG_enable LOG_reset

**LOG\_enable***Enable a message log***C Interface****Syntax**                      LOG\_enable(log);**Parameters**                LOG\_Handle log;            /\* log object handle \*/**Return Value**              Void**Reentrant**                      no**Description**                LOG\_enable enables the logging mechanism and allows the log buffer to be modified.**Example**                        LOG\_enable(&trace);**See Also**                      LOG\_disable  
LOG\_reset



**LOG\_error***Write an error message to the system log***C Interface**

**Syntax** `LOG_error(format, arg0);`

**Parameters**      String      format;      /\* printf-style format string \*/  
                      Arg          arg0;      /\* copied to second word of log record \*/

**Return Value**      Void

**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.

**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\_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
<b>Reentrant</b>	yes
<b>Description</b>	LOG_event copies a sequence number and three arguments to the specified log buffer. Each log message uses four words. The contents of the four words written by LOG_event are shown here:



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 Tconf to choose raw data for the datatype property and type a format string for the format property (see “LOG Object Properties” on page 2-215).

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,  
          (Arg)CLK_gettime());
```

**See Also**

- LOG\_error
- LOG\_printf
- TRC\_disable
- TRC\_enable

**LOG\_event5***Append an unformatted 5-argument message to a message log***C Interface****Syntax**

LOG\_event5(log, arg0, arg1, arg2, arg3, arg4);

**Parameters**

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 \*/  
 Arg       arg3;       /\* copied to fifth word of log record \*/  
 Arg       arg4;       /\* copied to sixth word of log record \*/

**Return Value**

Void

**Reentrant**

yes

**Description**

LOG\_event5 copies a sequence number, timestamp, and up to five arguments to the specified log buffer.

In order to use LOG\_event5, you *must* have configured the LOG.TS (timestamped logs) property to be true. The default is false. See LOG Manager Properties for details.

If you enable timestamped logs, you cannot view log data with the DSP/BIOS plug-ins in CCStudio. You can access them with the System Analyzer tool.

When you enable timestamped LOGs, each log record contains eight words. The contents of the eight words written by LOG\_event5 are shown here:



You can use a combination of calls to LOG\_event, LOG\_event5, LOG\_printf, and LOG\_printf4 to the same log so long as timestamped logs are enabled.

If you want the Event Log to apply the same printf-style format string to all records in the log, use Tconf to choose raw data for the datatype property and type a format string for the format property (see “LOG Object Properties” on page 2-215).

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\_event5. Log messages are never lost due to thread preemption.

**Example**

```
LOG_event5(&trace, (Arg)value0, (Arg)value1, (Arg)value2,  
           (Arg)value3, (Arg)CLK_gettime());
```

**See Also**

LOG\_event  
LOG\_printf4

**LOG\_message***Write a program-supplied message to the system log***C Interface**

**Syntax** `LOG_message(format, arg0);`

**Parameters**      String      format;      /\* printf-style format string \*/  
                      Arg          arg0;      /\* copied to second word of log record \*/

**Return Value**      Void

**Reentrant**          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 2-224, for details.

**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\_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
Reentrant	yes
Description	<p>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.</p> <p>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.</p> <p>The arguments passed to LOG_printf must be integers, strings, or a pointer (if the special %r or %p conversion character is used).</p> <p>The format string can use any conversion character found in Table 2-5.</p>

Table 2-5. Conversion Characters for LOG\_printf

Conversion Character	Description
%d	Signed integer
%u	Unsigned integer
%x	Unsigned hexadecimal integer
%o	Unsigned octal integer

Conversion Character	Description
%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>
%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	pointer

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

Each log message uses four words. The contents of the message written by LOG\_printf are shown here:

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

You configure the characteristics of a log in Tconf. 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 supports only 0, 1, or 2 arguments after the format string.
- ❑ The format string address is put in b6 as the third value for LOG\_event.

**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\_printf4**

*Append a formatted message with up to 4 arguments to message log*

**C Interface****Syntax**

```
LOG_printf4(log, format, arg0, arg1, arg2, arg3);
```

**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 */
Arg         arg2;    /* value for third format string token */
Arg         arg3;    /* value for fourth format string token */
```

**Return Value**

Void

**Reentrant**

yes

**Description**

This variant on the LOG\_printf function allows you to provide up to 4 arguments to be formatted by the format string. Four arguments are not required—you may use 0, 1, 2, 3, or 4 arguments.

This function behaves the same as LOG\_printf with the following exceptions:

LOG\_printf4 copies a sequence number, format address, timestamp, and up to four arguments to the specified log buffer.

In order to use LOG\_printf4, you *must* have configured the LOG.TS (timestamped logs) property to be true. The default is false. See LOG Manager Properties for details.

If you enable timestamped logs, you cannot view log data with the DSP/BIOS plug-ins in CCStudio. You can access them with the System Analyzer tool.

When you enable timestamped LOGs, each log record contains eight words. The contents of the eight words written by LOG\_printf4 are shown here:



You can use a combination of calls to LOG\_event, LOG\_event5, LOG\_printf, and LOG\_printf4 to the same log so long as timestamped logs are enabled.

**Constraints and  
Calling Context**

❑ none

**Example**

```
LOG_printf4(&trace, "hello world");  
LOG_printf4(&trace, "Data: %d %d %d %d", data1, data2,  
data3, data4);
```

**See Also**

LOG\_event5  
LOG\_printf

**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

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

<b>Description</b>	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.

<b>Example</b>	LOG_reset(&trace);
----------------	--------------------

<b>See Also</b>	LOG_disable LOG_enable
-----------------	---------------------------

## 2.16 MBX Module

Functions

The MBX module is the mailbox manager.

- ❑ 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,
};
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the MBX Manager Properties and MBX Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("IDRAM")

Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
messageSize	Int16	1
length	Int16	1
elementSeg	Reference	prog.get("IDRAM")

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 Tconf.

`MBX_pend` waits for a message from a mailbox. Its timeout parameter 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 DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment that contains the MBX objects created with Tconf.

Tconf Name: OBJMEMSEG                      Type: Reference

Example: `bios.MBX.OBJMEMSEG = prog.get("myMEM");`

## MBX Object Properties

To create an MBX object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myMbx = bios.MBX.create("myMbx");
```

The following properties can be set for an MBX object in the MBX Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: comment	Type: String
---------------------	--------------

Example: `myMbx.comment = "my MBX";`

- ❑ **Message Size.** The size (in MADUs, 8-bit bytes) of the messages this mailbox can contain.

Tconf Name: messageSize                      Type: Int16

Example: `myMbx.messageSize = 1;`

- ❑ **Mailbox Length.** The number of messages this mailbox can contain.

Tconf Name: length Type: Int16

Example: `myMbx.length = 1;`

- ❑ **Element memory segment.** The memory segment to contain the mailbox data buffers.

Tconf Name: elementSeg                      Type: Reference

Example: `myMbx.elementSeq = prog.get("myMEM");`

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

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

**Parameters**

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

**Return Value**

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

**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 2-240.

**Constraints and  
Calling Context**

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

**See Also**

```
MBX_delete
SYS_error
```

**MBX\_delete***Delete a mailbox***C Interface**

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

**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 a 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 Tconf, SYS\_error is called.

**See Also**

MBX\_create

**MBX\_pend***Wait for a message from mailbox***C Interface****Syntax**

```
status = MBX_pend(mbx, msg, timeout);
```

**Parameters**

```
MBX_Handle mbx;    /* mailbox object handle */
Ptr          msg;   /* message pointer */
Uns          timeout; /* return after this many system clock ticks */
```

**Return Value**

```
Bool          status; /* TRUE if successful, FALSE if timeout */
```

**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**

- ❑ This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- ❑ If you need to call MBX\_pend within a TSK\_disable/TSK\_enable block, you must use a timeout of 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****Syntax**

```
status = MBX_post(mbx, msg, timeout);
```

**Parameters**

```
MBX_Handle mbx;    /* mailbox object handle */
Ptr         msg;    /* message pointer */
Uns         timeout; /* return after this many system clock ticks */
```

**Return Value**

```
Bool         status; /* TRUE if successful, FALSE if timeout */
```

**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**

- ❑ If you need to call MBX\_post within a TSK\_disable/TSK\_enable block, you must use a timeout of 0.
- ❑ This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- ❑ 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

## 2.17 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\_getBaseAddress. Get base address of memory heap.
- ❑ MEM\_increaseTableSize. Increase the internal MEM table size.
- ❑ MEM\_redefine. Redefine an existing memory segment.
- ❑ MEM\_stat. Return the status of a memory segment.
- ❑ MEM\_undefine. Undefine an existing 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 */
    MEM_sizep length;   /* size of the segment */
    Uns      space;     /* memory space */
} MEM_Segment;

typedef struct MEM_Stat {
    MEM_sizep size;     /* original size of segment */
    MEM_sizep used;     /* MADUs used in segment */
    size_t    length;   /* largest contiguous block */
} MEM_Stat;

typedef unsigned int  MEM_sizep;
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. The defaults shown are for 'C62x and 'C67x. The memory segment defaults are different for 'C64x. For details, see the MEM Manager Properties and MEM Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters.**

<b>Name</b>	<b>Type</b>	<b>Default (Enum Options)</b>
REUSECODESPACE	Bool	false
ARGSSIZE	Numeric	0x0004
STACKSIZE	Numeric	0x0100
NOMEMORYHEAPS	Bool	false
BIOSOBJSEG	Reference	prog.get("IDRAM")
MALLOCSEG	Reference	prog.get("IDRAM")
USEMPC	Bool	false (C64x+ only)
ARGSSEG	Reference	prog.get("IDRAM")
STACKSEG	Reference	prog.get("IDRAM")
GBLINITSEG	Reference	prog.get("IDRAM")
TRCDATASEG	Reference	prog.get("IDRAM")
SYSDATASEG	Reference	prog.get("IDRAM")
OBJSEG	Reference	prog.get("IDRAM")
BIOSSEG	Reference	prog.get("IPRAM")
SYSINITSEG	Reference	prog.get("IPRAM")
HWISEG	Reference	prog.get("IPRAM")
HWIVCSEG	Reference	prog.get("IPRAM")
RTDXTEXTSEG	Reference	prog.get("IPRAM")
USERCOMMANDFILE	Bool	false
TEXTSEG	Reference	prog.get("IPRAM")
SWITCHSEG	Reference	prog.get("IDRAM")
BSSSEG	Reference	prog.get("IDRAM")
FARSEG	Reference	prog.get("IDRAM")
CINITSEG	Reference	prog.get("IDRAM")
PINITSEG	Reference	prog.get("IDRAM")
CONSTSEG	Reference	prog.get("IDRAM")
DATASEG	Reference	prog.get("IDRAM")
CIOSEG	Reference	prog.get("IDRAM")
ENABLELOADADDR	Bool	false
LOADBIOSSEG	Reference	prog.get("IPRAM")
LOADSYSINITSEG	Reference	prog.get("IPRAM")
LOADGBLINITSEG	Reference	prog.get("IDRAM")
LOADTRCDATASEG	Reference	prog.get("IDRAM")
LOADTEXTSEG	Reference	prog.get("IPRAM")

Name	Type	Default (Enum Options)
LOADSWITCHSEG	Reference	prog.get("IDRAM")
LOADCINITSEG	Reference	prog.get("IDRAM")
LOADPINITSEG	Reference	prog.get("IDRAM")
LOADCONSTSEG	Reference	prog.get("IDRAM")
LOADHWISEG	Reference	prog.get("IPRAM")
LOADHWIVECSEG	Reference	prog.get("IPRAM")
LOADRTDXTEXTSEG	Reference	prog.get("IPRAM")

### Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
base	Numeric	0x00000000
len	Numeric	0x00000000
createHeap	Bool	true
heapSize	Numeric	0x08000
enableHeapLabel	Bool	false
heapLabel	Extern	prog.extern("segment_name", "asm")
space	EnumString	"data" ("code", "code/data")

### 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 Tconf.

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, set the "No Dynamic Memory Heaps" property for the MEM manager to true. If you remove this capability, your program cannot call any of the MEM functions or any object creation functions (such as TSK\_create). You need to create all objects to be used by your program statically (with Tconf). You can also create or remove the dynamic memory heap from an individual memory segment in the configuration.

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.

## MEM Manager Properties

The following global properties can be set for the MEM module in the MEM Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

## General tab

- Tconf Name: REUSECODESPACE Type: Bool

Type: Bool

Example: `bios.MEM.REUSECODESPACE = false;`

- Tconf Name: ARGSSIZE                      Type: Numeric

Type: Numeric

Example: `bios.MEM.ARGSSIZE = 0x0004;`

- ❑ **Stack Size.** The size of the global stack in MADUs. The upper-left corner of the DSP/BIOS 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 c6000 this is an 8-bit byte.

Tconf Name: STACKSIZE

Type: Numeric

Example: `bios.MEM.STACKSIZE = 0x0400;`

- ❑ **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 property is set to true, the 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 property is set to true, the Segment For DSP/BIOS Objects, Segment for malloc()/free(), and Stack segment for dynamic tasks properties are set to MEM\_NULL.

When you set this property to true, heaps already specified in MEM segments are removed from the configuration. If you later reset this property to false, recreate heaps by configuring properties for individual MEM objects as needed.

Tconf Name: NOMEMORYHEAPS

Type: Bool

Example: `bios.MEM.NOMEMORYHEAPS = false;`

- ❑ **Segment For DSP/BIOS Objects.** The default memory segment to 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.

Tconf Name: BIOSOBJSEG

Type: Reference

Example: `bios.MEM.BIOSOBJSEG = prog.get("myMEM");`

- ❑ **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.

Tconf Name: MALLOCSEG

Type: Reference

Example: `bios.MEM.MALLOCSEG = prog.get("myMEM");`

- ❑ **Enable Memory Protection Controller module.** C64x+ only. Set this property to true to enable the MPC Module. By default, this module is disabled. If you plan to use the MPC module, you should also leave the EXC module enabled. This is set in the "Enable EXC module exception processing" field in the HWI Manager Properties.

Tconf Name: USEMPC

Type: Bool

Example: `bios.MEM.USEMPC = false;`

## BIOS Data tab

- ❑ **Argument Buffer Section (.args).** The memory segment containing the .args section.

Tconf Name: ARGSSEG

Type: Reference

Example: `bios.MEM.ARGSSEG = prog.get("myMEM");`

- ❑ **Stack Section (.stack).** The memory segment containing the global stack. This segment should be located in RAM.

Tconf Name: STACKSEG

Type: Reference

Example: `bios.MEM.STACKSEG = prog.get("myMEM");`

- ❑ **DSP/BIOS Init Tables (.gblinit).** The memory segment containing the DSP/BIOS global initialization tables.

Tconf Name: GBLINITSEG

Type: Reference

Example: `bios.MEM.GBLINITSEG = prog.get("myMEM");`

- ❑ **TRC Initial Value (.trcdata).** The memory segment containing the TRC mask variable and its initial value. This segment must be placed in RAM.

Tconf Name: TRCDATASEG

Type: Reference

Example: `bios.MEM.TRCDATASEG = prog.get("myMEM");`

- ❑ **DSP/BIOS Kernel State (.sysdata).** The memory segment containing system data about the DSP/BIOS kernel state.

Tconf Name: SYSDATASEG

Type: Reference

Example: `bios.MEM.SYSDATASEG = prog.get("myMEM");`

- ❑ **DSP/BIOS Conf Sections (.obj).** The memory segment containing configuration properties that can be read by the target program.

Tconf Name: OBJSEG

Type: Reference

Example: `bios.MEM.OBJSEG = prog.get("myMEM");`

## BIOS Code tab

- ❑ **BIOS Code Section (.bios).** The memory segment containing the DSP/BIOS code.

Tconf Name: BIOSSEG

Type: Reference

Example: `bios.MEM.BIOSSEG = prog.get("myMEM");`

- ❑ **Startup Code Section (.sysinit).** The memory segment containing DSP/BIOS startup initialization code; this memory can be reused after main starts executing.

Tconf Name: SYSINITSEG

Type: Reference

Example: `bios.MEM.SYSINITSEG = prog.get("myMEM");`

- ❑ **Function Stub Memory (.hwi).** The memory segment containing dispatch code for HWIs that are configured to be monitored in the HWI Object Properties.

Tconf Name: HWISEG

Type: Reference

Example: `bios.MEM.HWISEG = prog.get("myMEM");`

- ❑ **Interrupt Service Table Memory (.hwi\_vec).** The memory segment containing the Interrupt Service Table (IST). The IST can be placed anywhere on the memory map, but a copy of the RESET vector always remains at address 0x00000000.

Tconf Name: HWIVCSEG

Type: Reference

Example: `bios.MEM.HWIVCSEG = prog.get("myMEM");`

- ❑ **RTDX Text Segment (.rtdx\_text).** The memory segment containing the code sections for the RTDX module.

Tconf Name: RTDXTEXTSEG

Type: Reference

Example: `bios.MEM.RTDXTEXTSEG =  
prog.get("myMEM");`

## Compiler Sections tab

- ❑ **User .cmd File For Compiler Sections.** Put a checkmark in this box if you want to have full control over the memory used for the sections that follow. You must then create a linker command file that begins by including the linker command file created by the configuration. Your linker command file should then assign memory for the items normally handled by the following properties. See the *TMS320C6000 Optimizing Compiler User's Guide* for more details.

Tconf Name: USERCOMMANDFILE

Type: Bool

Example: `bios.MEM.USERCOMMANDFILE = false;`

- ❑ **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.

Tconf Name: TEXTSEG

Type: Reference

Example: `bios.MEM.TEXTSEG = prog.get("myMEM");`



- ❑ **Switch Jump Tables (.switch).** The memory segment containing the jump tables for switch statements. This segment can be located in ROM or RAM.

Tconf Name: SWITCHSEG

Type: Reference

Example: `bios.MEM.SWITCHSEG = prog.get("myMEM");`

- ❑ **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.

Tconf Name: BSSSEG

Type: Reference

Example: `bios.MEM.BSSSEG = prog.get("myMEM");`

- ❑ **C Variables Section (.far).** The memory segment containing global and static variables declared as far variables.

Tconf Name: FARSEG

Type: Reference

Example: `bios.MEM.FARSEG = prog.get("myMEM");`

- ❑ **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.

Tconf Name: CINITSEG

Type: Reference

Example: `bios.MEM.CINITSEG = prog.get("myMEM");`

- ❑ **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.

Tconf Name: PINITSEG

Type: Reference

Example: `bios.MEM.PINITSEG = prog.get("myMEM");`

- ❑ **Constant Sections (.const, .printf).** These sections can be located in ROM or RAM. The .const section contains string constants and data defined with the const C qualifier. The DSP/BIOS .printf section contains other constant strings used by the Real-Time Analysis tools. The .printf section is not loaded onto the target. Instead, the (COPY) directive is used for this section in the .cmd file. The .printf section is managed along with the .const section, since it must be grouped with the .const section to make sure that no addresses overlap. If you specify these sections in your own .cmd file, you'll need to do something like the following:

```
GROUP {
    .const: {}
    .printf (COPY): {}
} > IRAM
```

Tconf Name: CONSTSEG Type: Reference

Example: `bios.MEM.CONSTSEG = prog.get("myMEM");`

- ☐ **Data Section (.data).** This memory segment contains program data. This segment can be located in ROM or RAM.

Tconf Name: DATASEG Type: Reference

Example: `bios.MEM.DATASEG = prog.get("myMEM");`

- ☐ **Data Section (.cio).** This memory segment contains C standard I/O buffers.

Tconf Name: CIOSEG Type: Reference

Example: `bios.MEM.CIOSEG = prog.get("myMEM");`

## 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.

Tconf Name: ENABLELOADADDR Type: Bool

Example: `bios.MEM.ENABLELOADADDR = false;`

- ☐ **Load Address - BIOS Code Section (.bios).** The memory segment containing the load allocation of the section that contains DSP/BIOS code.

Tconf Name: LOADBIOSSSEG Type: Reference

Example: `bios.MEM.LOADBIOSSSEG =  
prog.get("myMEM");`

- ```
Example: bios.MEM.LOADCINITSEG =
         prog.get("myMEM");
```

- ❑ **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.

Tconf Name: LOADPINITSEG                      Type: Reference

```
Example:  bios.MEM.LOADPINITSEG =
          prog.get("myMEM");
```

- ❑ **Load Address - Constant Sections (.const, .printf).** The memory segment containing the load allocation of the sections that contain string constants, data defined with the const C qualifier, and other constant strings used by the Real-Time Analysis tools. The .printf section is managed along with the .const section to make sure that no addresses overlap.

Tconf Name: LOADCONSTSEG                      Type: Reference

```
Example:  bios.MEM.LOADCONSTSEG =
          prog.get("myMEM");
```

- ❑ **Load Address - Function Stub Memory (.hwi).** The memory segment containing the load allocation of the section that contains dispatch code for HWIs configured to be monitored.

Tconf Name: LOADHWISEG                      Type: Reference

Example: `bios.MEM.LOADHWISEG = prog.get("myMEM");`

- ❑ **Load Address - Interrupt Service Table Memory (.hwi\_vec).** The memory segment containing the load allocation of the section that contains the Interrupt Service Table (IST).

Tconf Name: LOADHWIVCSEG Type: Reference

```
Example:  bios.MEM.LOADHWIVECSEG =
          prog.get ("myMEM") ;
```

- ❑ **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.

Tconf Name: LOADRTDXTEXTSEG      Type: Reference

```
Example: bios.MEM.LOADRTDXTEXTSEG =  
         prog.get ("myMEM" );
```

## MEM Object Properties

A memory segment represents a contiguous length of code or data memory in the address space of the processor.

To create a MEM object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myMem = bios.MEM.create("myMem");
```

The following properties can be set for a MEM object in the MEM Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this MEM object.  
 Tconf Name: `comment` Type: String  
 Example: `myMem.comment = "my MEM";`
- ❑ **base.** The address at which this memory segment begins. This value is shown in hex.  
 Tconf Name: `base` Type: Numeric  
 Example: `myMem.base = 0x00000000;`
- ❑ **len.** The length of this memory segment in MADUs. This value is shown in hex.  
 Tconf Name: `len` Type: Numeric  
 Example: `myMem.len = 0x00000000;`
- ❑ **create a heap in this memory.** If this property is set to true, 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.  
 Tconf Name: `createHeap` Type: Bool  
 Example: `myMem.createHeap = true;`
- ❑ **heap size.** The size of the heap in MADUs to be created in this memory segment. You cannot control the location of the heap within its memory segment except by making the segment and heap the same sizes. Note that if the base of the heap ends up at address 0x0, the base address of the heap is offset by MEM\_HEADERSIZE and the heap size is reduced by MEM\_HEADERSIZE.  
 Tconf Name: `heapSize` Type: Numeric  
 Example: `myMem.heapSize = 0x08000;`
- ❑ **enter a user defined heap identifier.** If this property is set to true, you can define your own identifier label for this heap.  
 Tconf Name: `enableHeapLabel` Type: Bool  
 Example: `myMem.enableHeapLabel = false;`

- ❑ **heap identifier label.** If the property above is set to true, type a name for this segment's heap.

Tconf Name: heapLabel Type: Extern

Example:     myMem.heapLabel =  
                                  prog.extern("seg\_name", "asm");

- ❑ **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.

Tconf Name: space Type: EnumString

Options:     "code", "data", "code/data"

Example:     myMem.space = "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 2-6 and Table 2-7 list segments that can be defined for the c6000:

Table 2-6.    *Typical Memory Segments for c6x EVM Boards*

| Name   | Memory Segment Type                 |
|--------|-------------------------------------|
| IPRAM  | Internal (on-device) program memory |
| IDRAM  | Internal (on-device) data memory    |
| SBSRAM | External SBSRAM on CE0              |
| SDRAM0 | External SDRAM on CE2               |
| SDRAM1 | External SDRAM on CE3               |

Table 2-7.    *Typical Memory Segment for c6711 DSK Boards*

| Name  | Memory Segment Type |
|-------|---------------------|
| SDRAM | External SDRAM      |

**MEM\_alloc***Allocate from a memory segment***C Interface****Syntax**

```
addr = MEM_alloc(segid, size, align);
```

**Parameters**

```
Int          segid;    /* memory segment identifier */
size_t       size;     /* block size in MADUs */
size_t       align;    /* block alignment */
```

**Return Value**

```
Void         *addr;    /* address of allocated block of memory */
```

**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 to allocate memory from. This identifier can be an integer or a memory segment name defined in the configuration. Files created by the configuration 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.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK\_pend and LCK\_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_alloc cannot be called from the context of a SWI or HWI. MEM\_alloc checks the context from which it is called. It calls SYS\_error and returns MEM\_ILLEGAL if it is called from the wrong context.

A number of other DSP/BIOS APIs call MEM\_alloc internally, and thus also cannot be called from the context of a SWI or HWI. See the "Function Callability Table" on page A-2 for a detailed list of calling contexts for each DSP/BIOS API.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_alloc cannot be called from a SWI or HWI.
- ☐ MEM\_alloc cannot be called if the TSK scheduler is disabled.
- ☐ align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

**See Also**

```
MEM_calloc
MEM_free
MEM_valloc
```

**MEM\_calloc***Allocate from a memory segment and set value to 0***C Interface****Syntax**

```
addr = MEM_calloc(segid, size, align)
```

**Parameters**

```
Int          segid;    /* memory segment identifier */
size_t       size;     /* block size in MADUs */
size_t       align;    /* block alignment */
```

**Return Value**

```
Void         *addr;    /* address of allocated block of memory */
```

**Description**

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. The files created by the configuration 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.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK\_pend and LCK\_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_calloc cannot be called from the context of a SWI or HWI.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_calloc cannot be called from a SWI or HWI.
- ☐ MEM\_calloc cannot be called if the TSK scheduler is disabled.
- ☐ 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
std.h and stdlib.h functions
```



**MEM\_define***Define a new memory segment***C Interface****Syntax**

segid = MEM\_define(base, length, attrs);

**Parameters**

Ptr            base;        /\* base address of new segment \*/  
MEM\_sizep    length;      /\* length (in MADUs) of new segment \*/  
MEM\_Attrs    \*attrs;      /\* segment attributes \*/

**Return Value**

Int            segid;       /\* ID of new segment \*/

**Reentrant**

yes

**Description**

MEM\_define defines a new memory segment for use by the DSP/BIOS MEM Module.

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.

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.

If there are undefined slots available in the internal table of memory segment identifiers, one of those slots is (re)used for the new segment. If there are no undefined slots available in the internal table, the table size is increased via MEM\_alloc. See MEM\_increaseTableSize to manage performance in this situation.

**Constraints and Calling Context**

- ❑ At least one segment must exist at the time MEM\_define is called.
- ❑ MEM\_define internally locks the memory by calling LCK\_pend and LCK\_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_define cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.

- ❑ The length parameter must be a multiple of MEM\_HEADERSIZE and must be at least equal to MEM\_HEADERSIZE.
- ❑ The base Ptr cannot be NULL.

**See Also**

MEM\_redefine  
MEM\_undefine

**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 \*/  
 size\_t        size;     /\* block length in MADUs\*/

**Return Value**

Bool           status;   /\* TRUE if successful \*/

**Description**

MEM\_free places the memory block specified by addr and size back into the free pool of the segment specified by segid. The newly freed block is combined with any adjacent free blocks. 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.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK\_pend and LCK\_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_free cannot be called from the context of a SWI or HWI.

**Constraints and Calling Context**

- ❑ addr must be a valid pointer returned from a call to MEM\_alloc.
- ❑ segid and size are those values used in a previous call to MEM\_alloc.
- ❑ MEM\_free cannot be called by HWI or SWI functions.
- ❑ MEM\_free cannot be called if the TSK scheduler is disabled.

**See Also**

MEM\_alloc  
 std.h and stdlib.h functions

**MEM\_getBaseAddress**

*Get base address of a memory heap*

**C Interface**

|                                        |                                                                                                                        |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | addr = MEM_getBaseAddress(segid);                                                                                      |
| <b>Parameters</b>                      | Int            segid;    /* memory segment identifier */                                                               |
| <b>Return Value</b>                    | Ptr            addr;    /* heap base address pointer */                                                                |
| <b>Description</b>                     | MEM_getBaseAddress returns the base address of the memory heap with the segment ID specified by the segid parameter.   |
| <b>Constraints and Calling Context</b> | <div><input type="checkbox"/> The segid can be an integer or a memory segment name defined in the configuration.</div> |
| <b>See Also</b>                        | MEM Object Properties                                                                                                  |

**MEM\_increaseTableSize***Increase the internal MEM table size***C Interface**

|                     |                                             |             |                                               |
|---------------------|---------------------------------------------|-------------|-----------------------------------------------|
| <b>Syntax</b>       | status = MEM_increaseTableSize(numEntries); |             |                                               |
| <b>Parameters</b>   | Uns                                         | numEntries; | /* number of segments to increase table by */ |
| <b>Return Value</b> | Int                                         | status;     | /* TRUE if successful */                      |

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

**Description**

MEM\_increaseTableSize allocates numEntries of undefined memory segments. When MEM\_define is called, undefined memory segments are re-used. If no undefined memory segments exist, one is allocated. By using MEM\_increaseTableSize, the application can avoid the use of MEM\_alloc (thus improving performance and determinism) within the MEM\_define call.

MEM\_increaseTableSize internally locks memory by calling LCK\_pend and LCK\_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_increaseTableSize cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.

MEM\_increaseTableSize returns SYS\_OK to indicate success and SYS\_EALLOC if an allocation error occurred.

**Constraints and Calling Context**

- ❑ Do not call from the context of a SWI or HWI.

**See Also**

MEM\_define  
MEM\_undefine

**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 \*/  
MEM\_sizep    length;   /\* length (in MADUs) of new block \*/

**Return Value**

Void

**Reentrant**

yes

**Description**

MEM\_redefine redefines an existing memory segment managed by the DSP/BIOS MEM Module. 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.

**Constraints and  
Calling Context**

- ❑ MEM\_redefine internally locks the memory by calling LCK\_pend and LCK\_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_redefine cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.
- ❑ The length parameter must be a multiple of MEM\_HEADERSIZE and must be at least equal to MEM\_HEADERSIZE.
- ❑ The base Ptr cannot be NULL.

**See Also**

MEM\_define  
MEM\_undefine

**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 \*/

**Description**

MEM\_stat returns the status of the memory segment specified by segid in the status structure pointed to by statbuf.

```
typedef struct MEM_Stat {
    MEM_sizep size; /* original size of segment */
    MEM_sizep used; /* MADUs used in segment */
    size_t length; /* largest contiguous block */
} MEM_Stat;
```

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. If the segment has been undefined with MEM\_undefine, this function returns FALSE.

MEM functions that access memory internally lock the memory by calling the LCK\_pend and LCK\_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_stat cannot be called from the context of a SWI or HWI.

**Constraints and Calling Context**

- ❑ MEM\_stat cannot be called from a SWI or HWI.
- ❑ MEM\_stat cannot be called if the TSK scheduler is disabled.

**MEM\_undefine**

*Undefine an existing memory segment*

**C Interface**

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | MEM_undefine(segid);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Parameters</b>                      | Int                segid;     /* segment to undefine */                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Return Value</b>                    | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b>                     | <p>MEM_undefine removes a memory segment from the internal memory tables. Once a memory segment has been undefined, the segid cannot be used in any of the MEM APIs (except MEM_stat). Note: The undefined segid might later be returned by a subsequent MEM_define call.</p> <p>MEM_undefine internally locks the memory by calling LCK_pend and LCK_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_undefine cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ Do not call from the context of a SWI or HWI.</li><li>❑ MEM_undefine does not free the actual memory buffer managed by the memory segment.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>See Also</b>                        | MEM_define<br>MEM_redefine                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |



**MEM\_valloc***Allocate from a memory segment and set value***C Interface****Syntax**

```
addr = MEM_valloc(segid, size, align, value);
```

**Parameters**

```
Int          segid;    /* memory segment identifier */
size_t       size;     /* block size in MADUs */
size_t       align;    /* block alignment */
Char         value;    /* character value */
```

**Return Value**

```
Void         *addr;    /* address of allocated block of memory */
```

**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. The files created by the configuration 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.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK\_pend and LCK\_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM\_valloc cannot be called from the context of a SWI or HWI.

**Constraints and Calling Context**

- ☐ segid must identify a valid memory segment.
- ☐ MEM\_valloc cannot be called from a SWI or HWI.
- ☐ MEM\_valloc cannot be called if the TSK scheduler is disabled.
- ☐ 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  
 std.h and stdlib.h functions

## 2.18 MPC Module

The MPC module is the Memory Protection Controller manager for C64x+ devices.

### Functions

- ❑ **MPC\_getPA.** Get permission attributes of address.
- ❑ **MPC\_getPageSize.** Get size of page containing address.
- ❑ **MPC\_getPrivMode.** Get current CPU privilege mode.
- ❑ **MPC\_setBufferPA.** Set permission attributes for a buffer.
- ❑ **MPC\_setPA.** Set permission attributes for an address.
- ❑ **MPC\_setPrivMode.** Set CPU privilege mode.

### Constants, Types, and Structures

```
/* MPPA Permission Attribute Register bitmasks */
typedef Uns MPC_Perm;

/* macros for valid values for MPC_Perm */
#define MPC_MPPA_UX 0x00000001 /* User eXecute */
#define MPC_MPPA_UW 0x00000002 /* User Write */
#define MPC_MPPA_UR 0x00000004 /* User Read */
#define MPC_MPPA_SX 0x00000008 /* Supervisor eXecute */
#define MPC_MPPA_SW 0x00000010 /* Supervisor Write */
#define MPC_MPPA_SR 0x00000020 /* Supervisor Read */
#define MPC_MPPA_LOCAL 0x00000100 /* LOCAL CPU */
```

### Configuration Properties

The MPC module has no configuration properties of its own. To enable the MPC module, set the “Enable Memory Protection Controller module” field in the MEM Manager Properties to true.

### Description

Memory protection can protect operating system data structures from poorly behaving code and aid in debugging by providing greater information about illegal memory accesses. The C64x+ Megamodule Memory Protection Architecture provides for memory protection through a combination of CPU privilege levels and a memory system permission structure.

- ❑ **CPU privilege levels.** Code running on the CPU executes in one of two privilege modes: Supervisor Mode or User Mode. The privilege of a thread determines what permissions that thread might have. You can use **MPC\_getPrivMode** and **MPC\_setPrivMode** to get and set the CPU privilege level.
  - **Supervisor code** is considered "more trusted" than User code. Examples of Supervisor threads include operating system kernels and hardware device drivers. Supervisor Mode is generally granted access to peripheral registers and the memory protection configuration.

- **User code** includes, for example, vocoders and end applications. User Mode is generally confined to the memory spaces that the OS specifically designates for its use.
- **Permission structure.** The Memory Protection model defines three fundamental functional access types: Read, Write, and Execute. Read and Write refer to data accesses by the CPU or the IDMA engine. Execute refers to accesses associated with a program fetch. You can enable/disable these access types on a per page basis for both User and Supervisor mode. Use `MPC_getPA`, `MPC_setBufferPA` and `MPC_setPA` to get and set the permissions.

See the chapter on "Memory Protection" in the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information.

### Handling Invalid Accesses

The MPC hardware generates exceptions when an access that violates permissions occurs. The DSP/BIOS MPC module is associated with an `"_MPC"` module (note the underscore) that contains code that reports permission violations.

When enabled, the MPC module assigns `_MPC` exception handling functions to the EXC exception handling hooks (see Section C.4, *EXC Module* and Section C.5, *\_MPC Module*). The MPC module enables and handles only MPC-related events.

If you use any MPC module APIs in your application source code, add the following line to your source file:

```
#include <mpc.h>
```

In addition to the MPC APIs, the `"_MPC"` APIs includes handler functions used with the EXC module hooks. Note the initial underscore in this module name. If you use any `_MPC` module APIs in your application source code, add the following line to your source file:

```
#include <_mpc.h>
```

Enabling the MPC module causes the Program Memory Controller (PMC), Data Memory Controller (DMC), and Unified Memory Controller (UMC) CPU events to be enabled to generate exceptions. The corresponding DMA events are not enabled.

If you want other exceptions to be generated, you need to enable those system events and write functions to handle them. For details, see the EXC Module. Since the MPC module takes control of the EXC function

hooks, the MPC module also provides a function hook that you can assign to handle additional exception processing (see `_MPC_userHook`).

When enabled, the MPC module assigns the following functions to the EXC function hooks.

- ☐ `EXC_exceptionHook = _MPC_exceptionHandler`
- ☐ `EXC_externalHook = _MPC_externalHandler`
- ☐ `EXC_internalHook = _MPC_internalHandler`
- ☐ `EXC_nmiHook = (not used)`

If you need to further process external exceptions, including ones already handled by the MPC module, you can write a function and assign it to the function pointer `_MPC_userHook`.

## **MPC Manager Properties**

By default, the MPC module is disabled. To enable it, set the “Enable Memory Protection Controller module” field in the MEM Manager Properties to true. You can also enable the MPC module in a Tconf script with the following statement:

```
bios.MEM.USEMPCC = true;
```

You can use MPC APIs without enabling the EXC Module, but you must have the EXC module enabled to allow MPC-related violations to generate an exception and report information about the exception. The EXC module is enabled by default.

Note that the MPC module does not have its own “module” in the configuration tools, and there are no “MPC objects”. It is a module in the DSP/BIOS kernel. The MEM module is used as a container for the single MPC configuration property.

**MPC\_getPA***Get permission attributes of address***C Interface****Syntax**

```
status = MPC_getPA(addr, space, *perm);
```

**Parameters**

```
Ptr      addr;    /* address to request permissions for */
Int      space;   /* memory space of addr */
MPC_Perm *perm;   /* pointer to storage for desired PA */
```

**Return Value**

```
Int      status   /* SYS_OK or SYS_EINVAL */
```

**Description**

This function is available only for C64x+ devices.

MPC\_getPA reads the permission attributes (PA) associated with the specified location.

The addr parameter specifies an address for which you want to know the permissions. Due to the page granularity of the PA mask, all memory locations contained in the page in which addr resides have the same permission attributes.

The space parameter identifies whether the address is in program, data, I/O, or other memory. Since the C64x+ has a single memory space, use zero (0) for this parameter. Other values may be supported in future versions of DSP/BIOS.

The perm parameter is a pointer to a bitmask of type MPC\_Perm. You can use the following constants to interpret the bitmask:

```
#define MPC_MPPA_UX  0x00000001  /* User eXecute */
#define MPC_MPPA_UW  0x00000002  /* User Write */
#define MPC_MPPA_UR  0x00000004  /* User Read */
#define MPC_MPPA_SX  0x00000008  /* Supervisor eXecute */
#define MPC_MPPA_SW  0x00000010  /* Supervisor Write */
#define MPC_MPPA_SR  0x00000020  /* Supervisor Read */
#define MPC_MPPA_LOCAL 0x00000100 /* LOCAL CPU */
```

This function returns SYS\_OK if the operation is successful and SYS\_EINVAL if the address and space you specify are invalid.

**Constraints and Calling Context**

❑ none

**See Also**

MPC\_setBufferPA  
MPC\_setPA

**MPC\_getPageSize***Get size of page containing address***C Interface****Syntax**

```
status = MPC_getPageSize(addr, space, *pageSize);
```

**Parameters**

```
Ptr      addr;      /* address to request page size for */
Int      space;     /* memory space of addr */
Uns      *pageSize; /* pointer to storage for desired page size */
```

**Return Value**

```
Int      status     /* SYS_OK or SYS_EINVAL */
```

**Description**

This function is available only for C64x+ devices.

MPC\_getPageSize returns the page size associated with the specified address.

The addr parameter specifies an address for which you want to know the page size.

The space parameter identifies whether the address is in program, data, I/O, or other memory. Since the C64x+ has a single memory space, use zero (0) for this parameter. Other values may be supported in future versions of DSP/BIOS.

The pageSize parameter is a pointer to a location that will receive the page size of the specified location. The page size is measured in MAUs (minimum addressable units).

This function returns SYS\_OK if the operation is successful and SYS\_EINVAL if the address and space you specify are invalid.

**Constraints and  
Calling Context**

❑ none

**See Also**

MPC\_setBufferPA

**MPC\_getPrivMode** *Get current CPU privilege mode***C Interface**

**Syntax** `privMode = MPC_getPrivMode(Void);`

**Parameters** Void

**Return Value** Uns `privMode /* MPC_SV or MPC_US */`

**Description**

This function is available only for C64x+ devices.

MPC\_getPrivMode returns the current CPU privilege mode. The return value matches one of the following constants:

- ☐ MPC\_SV. Supervisor mode
- ☐ MPC\_US. User mode
- ☐ none

**Constraints and Calling Context**

**See Also** MPC\_setPrivMode

**MPC\_setBufferPA***Set permission attributes for a buffer***C Interface**

|                     |                                                                                                                                                                                                                                                                                                                                   |     |                                                               |     |                                    |     |                                       |          |                                          |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|---------------------------------------------------------------|-----|------------------------------------|-----|---------------------------------------|----------|------------------------------------------|
| <b>Syntax</b>       | <code>status = MPC_setBufferPA(baseAddr, size, space, perm);</code>                                                                                                                                                                                                                                                               |     |                                                               |     |                                    |     |                                       |          |                                          |
| <b>Parameters</b>   | <table> <tr> <td>Ptr</td><td>baseAddr; /* base address of buffer to set permissions for */</td></tr> <tr> <td>Uns</td><td>size; /* size in MAUs of buffer */</td></tr> <tr> <td>Int</td><td>space; /* memory space of baseAddr */</td></tr> <tr> <td>MPC_Perm</td><td>perm; /* permission attributes to set */</td></tr> </table> | Ptr | baseAddr; /* base address of buffer to set permissions for */ | Uns | size; /* size in MAUs of buffer */ | Int | space; /* memory space of baseAddr */ | MPC_Perm | perm; /* permission attributes to set */ |
| Ptr                 | baseAddr; /* base address of buffer to set permissions for */                                                                                                                                                                                                                                                                     |     |                                                               |     |                                    |     |                                       |          |                                          |
| Uns                 | size; /* size in MAUs of buffer */                                                                                                                                                                                                                                                                                                |     |                                                               |     |                                    |     |                                       |          |                                          |
| Int                 | space; /* memory space of baseAddr */                                                                                                                                                                                                                                                                                             |     |                                                               |     |                                    |     |                                       |          |                                          |
| MPC_Perm            | perm; /* permission attributes to set */                                                                                                                                                                                                                                                                                          |     |                                                               |     |                                    |     |                                       |          |                                          |
| <b>Return Value</b> | Int status /* SYS_OK or SYS_EINVAL */                                                                                                                                                                                                                                                                                             |     |                                                               |     |                                    |     |                                       |          |                                          |

**Description**

This function is available only for C64x+ devices.

MPC\_setBufferPA writes specified permission attributes for the specified buffer.

The baseAddr parameter specifies the start of an address for which you want to set the permissions.

The size parameter specifies the length of the buffer in MAUs. Due to the page granularity of the PA, memory locations not contained in the buffer but which exist on the same page as the beginning or end of the buffer are set with the same permission attributes. Using a size of 1 is equivalent to calling MPC\_setPA. You can find the page size for a particular address by calling MPC\_getPageSize.

The space parameter identifies whether the address is in program, data, I/O, or other memory. Since the C64x+ has a single memory space, use zero (0) for this parameter. Other values may be supported in future versions of DSP/BIOS.

The perm parameter is a bitmask of type MPC\_Perm. You can set any number of bits in the PA mask. You can use the following constants to set the bitmask:

```
#define MPC_MPPA_UX 0x00000001 /* User eXecute */
#define MPC_MPPA_UW 0x00000002 /* User Write */
#define MPC_MPPA_UR 0x00000004 /* User Read */
#define MPC_MPPA_SX 0x00000008 /* Supervisor eXecute */
#define MPC_MPPA_SW 0x00000010 /* Supervisor Write */
#define MPC_MPPA_SR 0x00000020 /* Supervisor Read */
#define MPC_MPPA_LOCAL 0x00000100 /* LOCAL CPU */
```

This function returns SYS\_OK if the operation is successful and SYS\_EINVAL if some or all of the buffer address range you specify is invalid.



**Constraints and  
Calling Context**☐ none**Example**

```
#define IRAM_CODE_BASE 0x00800000
#define IRAM_CODE_LEN 0x00008000

MPC_Perm perm;

/* Set code space to execute-only (user & supervisor) */
perm = MPC_MPPA_UX | MPC_MPPA_SX | MPC_MPPA_LOCAL;

MPC_setBufferPA((Ptr)IRAM_CODE_BASE, IRAM_CODE_LEN,
                0, perm);
```

**See Also**

MPC\_getPA  
MPC\_getPageSize  
MPC\_setPA

**MPC\_setPA***Set permission attributes for an address***C Interface****Syntax**

```
status = MPC_setPA(addr, space, perm);
```

**Parameters**

```
Ptr      addr;    /* address to set permissions for */
Int      space;   /* memory space of addr */
MPC_Perm perm;    /* permission attributes to set */
```

**Return Value**

```
Int      status   /* SYS_OK or SYS_EINVAL */
```

**Description**

This function is available only for C64x+ devices.

MPC\_setPA sets the permission attributes (PA) associated with the specified location.

The addr parameter specifies the address for which you want to set the permissions.

The space parameter identifies whether the address is in program, data, I/O, or other memory. Since the C64x+ has a single memory space, use zero (0) for this parameter. Other values may be supported in future versions of DSP/BIOS.

The perm parameter is a bitmask of type MPC\_Perm. You can set any number of bits in the PA mask. Due to the page granularity of the PA mask, all memory locations contained in the page in which addr resides are set with perm. You can use the following constants to set the bitmask:

```
#define MPC_MPPA_UX  0x00000001    /* User eXecute */
#define MPC_MPPA_UW  0x00000002    /* User Write */
#define MPC_MPPA_UR  0x00000004    /* User Read */
#define MPC_MPPA_SX  0x00000008    /* Supervisor eXecute */
#define MPC_MPPA_SW  0x00000010    /* Supervisor Write */
#define MPC_MPPA_SR  0x00000020    /* Supervisor Read */
#define MPC_MPPA_LOCAL 0x00000100    /* LOCAL CPU */
```

This function returns SYS\_OK if the operation is successful and SYS\_EINVAL if the address and space you specify are invalid.

To set permissions for a range of addresses, use the MPC\_setBufferPA API, instead.

**Constraints and Calling Context**

□ none

**See Also**

MPC\_getPA  
MPC\_setBufferPA

## **MPC\_setPrivMode** *Set CPU privilege mode*

### **C Interface**

|                     |                                                        |
|---------------------|--------------------------------------------------------|
| <b>Syntax</b>       | <code>MPC_setPrivMode(privMode);</code>                |
| <b>Parameters</b>   | Uns <code>privMode; /* privilege mode to set */</code> |
| <b>Return Value</b> | Void                                                   |

### **Description**

This function is available only for C64x+ devices.

MPC\_setPrivMode modifies the current CPU privilege mode. You can set the mode using one of the following constants:

- ☐ MPC\_SV. Supervisor mode
- ☐ MPC\_US. User mode

MPC\_setPrivMode relies on support by the EXC\_dispatch function, so the EXC Module must be enabled. The source code fore EXC\_dispatch is provided with DSP/BIOS in the exc\_asm.s64P file.

### **Constraints and Calling Context**

- ☐ none

### **Example**

```
/* temporarily set privilege mode to permit access */
MPC_setPrivMode(MPC_SV);
ptr = MEM_alloc(L1D_HEAP, 0x100, 0);
MPC_setPrivMode(MPC_US);
```

### **See Also**

MPC\_getPrivMode

## 2.19 MSGQ Module

The MSGQ module allows for the structured sending and receiving of variable length messages. This module can be used for homogeneous or heterogeneous multi-processor messaging.

### Functions

- ❑ MSGQ\_alloc. Allocate a message. Performed by writer.
- ❑ MSGQ\_close. Closes a message queue. Performed by reader.
- ❑ MSGQ\_count. Return the number of messages in a message queue.
- ❑ MSGQ\_free. Free a message. Performed by reader.
- ❑ MSGQ\_get. Receive a message from the message queue. Performed by reader.
- ❑ MSGQ\_getAttrs. Returns the attributes of a local message queue.
- ❑ MSGQ\_getDstQueue. Get destination message queue.
- ❑ MSGQ\_getMsgId. Return the message ID from a message.
- ❑ MSGQ\_getMsgSize. Return the message size from a message.
- ❑ MSGQ\_getSrcQueue. Extract the reply destination from a message.
- ❑ MSGQ\_isLocalQueue. Returns TRUE if local message queue.
- ❑ MSGQ\_locate. Synchronously find a message queue. Performed by writer.
- ❑ MSGQ\_locateAsync. Asynchronously find a message queue. Performed by writer.
- ❑ MSGQ\_open. Opens a message queue. Performed by reader.
- ❑ MSGQ\_put. Place a message on a message queue. Performed by writer.
- ❑ MSGQ\_release. Release a located message queue. Performed by writer.
- ❑ MSGQ\_setErrorHandler. Set up handling of internal MSGQ errors.
- ❑ MSGQ\_setMsgId. Sets the message ID in a message.
- ❑ MSGQ\_setSrcQueue. Sets the reply destination in a message.

### Constants, Types, and Structures

```
/* Attributes used to open message queue */
typedef struct MSGQ_Attrs {
    Ptr        notifyHandle;
    MSGQ_Pend  pend;
    MSGQ_Post  post;
} MSGQ_Attrs;
```

```

MSGQ_Attrs MSGQ_ATTRS = {
    NULL, /* notifyHandle */
    (MSGQ_Pend)SYS_zero, /* NOP pend */
    FXN_F_nop /* NOP post */
};

/* Attributes for message queue location */
typedef struct MSGQ_LocateAttrs {
    Uns timeout;
} MSGQ_LocateAttrs;

MSGQ_LocateAttrs MSGQ_LOCATEATTRS = {SYS_FOREVER};

/* Attrs for asynchronous message queue location */
typedef struct MSGQ_LocateAsyncAttrs {
    Uint16 poolId;
    Arg arg;
} MSGQ_LocateAttrs;

MSGQ_LocateAsyncAttrs MSGQ_LOCATEASYNCATTRS = {0, 0};

/* Configuration structure */
typedef struct MSGQ_Config {
    MSGQ_Obj *msgqQueues; /* Array of MSGQ handles */
    MSGQ_TransportObj *transports; /* Transport array */
    Uint16 numMsgqQueues; /* Number of MSGQ handles */
    Uint16 numProcessors; /* Number of processors */
    Uint16 startUninitialized; /* 1st MSGQ to init */
    MSGQ_Queue errorQueue; /* Receives transport err */
    Uint16 errorPoolId; /* Alloc errors from poolId */
} MSGQ_Config;

/* Asynchronous locate message */
typedef struct MSGQ_AsyncLocateMsg {
    MSGQ_MsgHeader header;
    MSGQ_Queue msgqQueue;
    Arg arg;
} MSGQ_AsyncLocateMsg;

/* Asynchronous error message */
typedef struct MSGQ_AsyncErrorMsg {
    MSGQ_MsgHeader header;
    MSGQ_MqtError errorType;
    Uint16 mqtId;
    Uint16 parameter;
} MSGQ_AsyncErrorMsg;

```

```
/* Transport object */
typedef struct MSGQ_TransportObj {
    MSGQ_MqtInit  initFxn;    /* Transport init func */
    MSGQ_TransportFxn *fxns; /* Interface funcs */
    Ptr          params; /* Setup parameters */
    Ptr          object; /* Transport-specific object */
    Uint16       procId; /* Processor Id talked to */
} MSGQ_TransportObj;
```

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the MSGQ Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

| Name       | Type | Default (Enum Options) |
|------------|------|------------------------|
| ENABLEMSGQ | Bool | false                  |

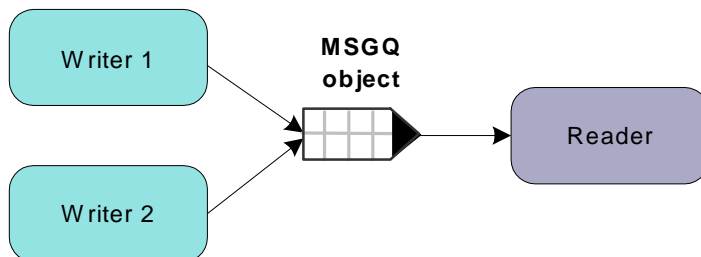
**Description**

The MSGQ module allows for the structured sending and receiving of variable length messages. This module can be used for homogeneous or heterogeneous multi-processor messaging. The MSGQ module with a substantially similar API is implemented in DSP/BIOS Link for certain TI general-purpose processors (GPPs), particularly those used in OMAP devices.

MSGQ provides more sophisticated messaging than other modules. It is typically used for complex situations such as multi-processor messaging. The following are key features of the MSGQ module:

- ❑ Writers and readers can be relocated to another processor with no runtime code changes.
- ❑ Timeouts are allowed when receiving messages.
- ❑ Readers can determine the writer and reply back.
- ❑ Receiving a message is deterministic when the timeout is zero.
- ❑ Sending a message is non-blocking.
- ❑ Messages can reside on any message queue.
- ❑ Supports zero-copy transfers.
- ❑ Can send and receive from HWIs, SWIs and TSKs.
- ❑ Notification mechanism is specified by application.
- ❑ Allows QoS (quality of service) on message buffer pools. For example, using specific buffer pools for specific message queues.

Messages are sent and received via a *message queue*. A reader is a thread that gets (reads) messages from a message queue. A writer is a thread that puts (writes) a message to a message queue. Each message queue has one reader and can have many writers. A thread may read from or write to multiple message queues.



*Figure 2-1. Writers and Reader of a Message Queue*

Conceptually, the reader thread owns a message queue. The processor where the reader resides opens a message queue. Writer threads locate existing message queues to get access to them.

Messages must be allocated from the MSGQ module. Once a message is allocated, it can be sent on any message queue. Once a message is sent, the writer loses ownership of the message and should not attempt to modify the message. Once the reader receives the message, it owns the message. It may either free the message or re-use the message.

Messages in a message queue can be of variable length. The only requirement is that the first field in the definition of a message must be a `MSGQ_MsgHeader` element.

```
typedef struct MyMsg {
    MSGQ_MsgHeader header;
    ...
} MyMsg;
```

The MSGQ API uses the `MSGQ_MsgHeader` internally. Your application should not modify or directly access the fields in the `MSGQ_MsgHeader`.

The MSGQ module has the following components:

- ❑ **MSGQ API.** Applications call the MSGQ functions to open and use a message queue object to send and receive messages. For an overview, see “MSGQ APIs” on page 2-274. For details, see the sections on the individual APIs.

- ❑ **Allocators.** Messages sent via MSGQ must be allocated by an allocator. The allocator determines where and how the memory for the message is allocated. For more about allocators, see the *DSP/BIOS User's Guide* (SPRU423F).
- ❑ **Transports.** Transports are responsible for locating and sending messages with other processors. For more about transports, see the *DSP/BIOS User's Guide* (SPRU423F).

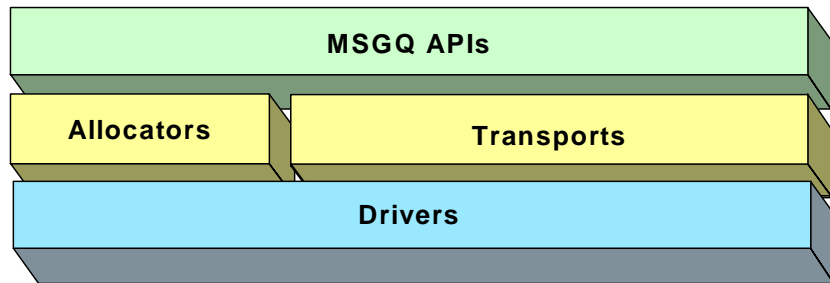


Figure 2-2. Components of the MSGQ Architecture

For more about using the MSGQ module—including information about multi-processor issues and a comparison of data transfer modules—see the *DSP/BIOS User's Guide* (SPRU423F).

## MSGQ APIs

The MSGQ APIs are used to open and close message queues and to send and receive messages. The MSGQ APIs shield the application from having to contain any knowledge about transports and allocators.

The following figure shows the call sequence of the main MSGQ functions:

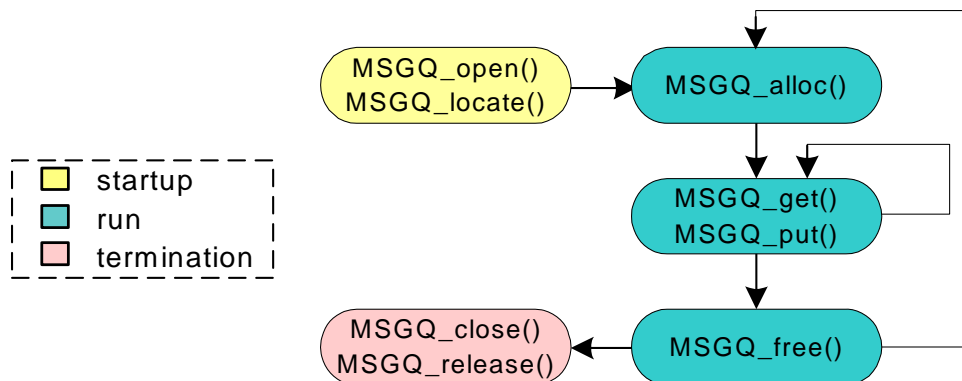


Figure 2-3. MSGQ Function Calling Sequence

The reader calls the following APIs:



- ❑ MSGQ\_open
- ❑ MSGQ\_get
- ❑ MSGQ\_free
- ❑ MSGQ\_close

A writer calls the following APIs:

- ❑ MSGQ\_locate or MSGQ\_locateAsync
- ❑ MSGQ\_alloc
- ❑ MSGQ\_put
- ❑ MSGQ\_release

Wherever possible, the MSGQ APIs have been written to have a deterministic execution time. This allows application designers to be certain that messaging will not consume an unknown number of cycles.

In addition, the MSGQ functions support use of message queues from all types of DSP/BIOS threads: HWIs, SWIs, and TSKs. That is, calls that may be synchronous (blocking) have an asynchronous (non-blocking) alternative.

## Static Configuration

In order to use the MSGQ module and the allocators it depends upon, you must statically configure the following:

- ❑ ENABLEMSGQ property of the MSGQ module using Tconf (see “MSGQ Manager Properties” on page 2-279)
- ❑ MSGQ\_config variable in application code (see below)
- ❑ PROCID property of the GBL module using Tconf (see “GBL Module Properties” on page 2-130)
- ❑ ENABLEPOOL property of the POOL module using Tconf (see “POOL Manager Properties” on page 2-330)
- ❑ POOL\_config variable in application code (see “Static Configuration” on page 2-327)

An application must provide a filled in MSGQ\_config variable in order to use the MSGQ module.

```
MSGQ_Config MSGQ_config;
```

The MSGQ\_Config type has the following structure:

```
typedef struct MSGQ_Config {
    MSGQ_Obj      *msgqQueues;      /* Array of message queue handles */
    MSGQ_TransportObj *transports;  /* Array of transports */
    UInt16        numMsgqQueues;    /* Number of message queue handles*/
    UInt16        numProcessors;    /* Number of processors */
    UInt16        startUninitialized; /* First msgq to init */
    MSGQ_Queue    errorQueue;       /* Receives async transport errors*/
    UInt16        errorPoolId;      /* Alloc error msgs from poolId */
} MSGQ_Config;
```

The fields in the MSGQ\_Config structure are described in the following table:

| Field              | Type                | Description                                                                                      |
|--------------------|---------------------|--------------------------------------------------------------------------------------------------|
| msgqQueues         | MSGQ_Obj *          | Array of message queue objects. The fields of each object do not need to be initialized.         |
| transports         | MSGQ_TransportObj * | Array of transport objects. The fields of each object must be initialized.                       |
| numMsgqQueues      | UInt16              | Length of the msgqQueues array.                                                                  |
| numProcessors      | UInt16              | Length of the transports array.                                                                  |
| startUninitialized | UInt16              | Index of the first message queue to initialize in the msgq-Queue array. This should be set to 0. |
| errorQueue         | MSGQ_Queue          | Message queue to receive transport errors. Initialize to MSGQ_INVALIDMSGQ.                       |
| errorPoolId        | UInt16              | Allocator to allocate transport errors. Initialize to POOL_INVALIDID.                            |

Internally, MSGQ references its configuration via the MSGQ\_config variable. If the MSGQ module is enabled (via Tconf) but the application does not provide the MSGQ\_config variable, the application cannot be linked successfully.

In the MSGQ\_Config structure, an array of MSGQ\_TransportObj items defines transport objects with the following structure:

```
typedef struct MSGQ_TransportObj {
    MSGQ_MqtInit  initFxn;    /* Transport init func */
    MSGQ_TransportFxn *fxns; /* Interface funcs */
    Ptr          params;     /* Setup parameters */
    Ptr          object;     /* Transport-specific object */
    UInt16       procId;     /* Processor Id talked to */
} MSGQ_TransportObj;
```

The following table describes the fields in the MSGQ\_TransportObj structure:

| Field   | Type                 | Description                                                                                                                                                                          |
|---------|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| initFxn | MSGQ_MqtlNit         | Initialization function for this transport. This function is called during DSP/BIOS startup. More explicitly it is called before main().                                             |
| fxns    | MSGQ_TransportFxns * | Pointer to the transport's interface functions.                                                                                                                                      |
| params  | Ptr                  | Pointer to the transport's parameters. This field is transport-specific. Please see documentation provided with your transport for a description of this field.                      |
| info    | Ptr                  | State information needed by the transport. This field is initialized and managed by the transport. Refer to the specific transport implementation to determine how to use this field |
| proclId | Uint16               | Numeric ID of the processor that this transport communicates with. The current processor must have a proclId field that matches the GBL.PROCID property.                             |

If no parameter structure is specified (that is, NULL is used) for the MSGQ\_TransportObj, the transport uses its default parameters.

The order of the transports array is by processor. The first entry communicates with processor 0, the next entry with processor 1, and so on. On processor n, the nth entry in the transport array should be MSGQ\_NOTTRANSPORT, since there is no transport to itself. The following example shows a configuration for a single-processor application (that is, processor 0). Note that the 0th entry is MSGQ\_NOTTRANSPORT

```
#define NUMMSGQUEUES 4 /* # of local message queues*/
#define NUMPROCESSORS 1 /* Single processor system */

static MSGQ_Obj      msgQueues [NUMMSGQUEUES] ;
static MSGQ_TransportObj transports [NUMPROCESSOR] =
                        {MSGQ_NOTTRANSPORT};

MSGQ_Config MSGQ_config = {
    msgQueues,
    transports,
    NUMMSGQUEUES,
    NUMPROCESSORS,
    0,
    MSGQ_INVALIDMSGQ,
    POOL_INVALIDIDID
};
```

## Managing Transports at Run-Time

As described in the previous section, MSGQ uses an array of transports of type MSGQ\_TransportObj in the MSGQ\_config variable. This array is processor ID based. For example, MSGQ\_config->transports[0] is the transport to processor 0. Therefore, if a single binary is used on multiple processors, the array must be changed at run-time.

As with the GBL\_setProclId API, the transports array can be managed in the User Init Function (see GBL Module Properties). DSP/BIOS only uses MSGQ\_config and the transports array after the User Init Function returns.

There are several ways to manage the transports array. Two common ways are as follows:

- ❑ **Create a static two-dimensional transports array and select the correct one.** Assume a single image will be used for two processors (proclId 0 and 1) in a system with NUMPROCESSORS (3 in this example) processors. The transports array in the single image might look like this:

```
MSGQ_TransportObj transports[2][NUMPROCESSORS] =
{ { MSGQ_NOTRANSPORT, // proc 0 talk to proc 0
  { ... },           // proc 0 talk to proc 1
  { ... },           // proc 0 talk to proc 2
},
  { { ... },          // proc 1 talk to proc 0
    MSGQ_NOTRANSPORT, // proc 1 talk to proc 1
    { ... },          // proc 1 talk to proc 2
  }
}
```

In the User Init Function, the application would call GBL\_setProclId with the correct processor ID. Then it would assign the correct transport array to MSGQ\_config. For example, for processor 1, it would do the following:

```
MSGQ_config.transports = transports[1];
```

Note that this approach does not scale well as the number of processors in the system increases.

- ❑ **Fill in the transports array in the User Init Function.** In the User Init Function, you can fill in the contents of the transports array. You would still statically define a 1-dimensional transports array as follows:

```
MSGQ_TransportObj transports[NUMPROCESSORS];
```

This array would not be initialized. The initialization would occur in the User Init Function. For example on processor 1, it would fill in the transports array as follows.

```

    transports[0].initFxn = ...
    transports[0].fxns = ...
    transports[0].object = ...
    transports[0].params = ...
    transports[0].procId = 0;
    transports[1] = MSGQ_NOTTRANSPORT;//no self-transport
    transports[2].initFxn = ...
    transports[2].fxns = ...
    ...
    transports[2].procId = 2;
    MSGQ_config.transport = transports;

```

Note that some of the parameters may not be able to be determined easily at run-time, therefore you may need to use a mixture of these two options.

### Message Queue Management

When a message queue is closed, the threads that located the closing message queue are not notified. No messages should be sent to a closed message queue. Additionally, there should be no active call to MSGQ\_get or MSGQ\_getAttrs to a message queue that is being closed. When a message queue is closed, all unread messages in the message queue are freed.

### MSGQ Manager Properties

To configure the MSGQ manager, the MSGQ\_Config structure must be defined in the C code. See “Static Configuration” on page 2-275.

The following global property must also be set in order to use the MSGQ module:

- ❑ **Enable Message Queue Manager.** If ENABLEMSGQ is TRUE, each transport and message queue specified in the MSGQ\_config structure (see “Static Configuration” on page 2-275) is initialized.

Tconf Name: ENABLEMSGQ

Type: Bool

Example:    bios.MSGQ.ENABLEMSGQ = true;

**MSGQ\_alloc***Allocate a message***C Interface**

|                     |                                                                                                                                                                                                                                                                                                       |                                                |         |                                                |          |       |                                       |        |       |                                     |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|---------|------------------------------------------------|----------|-------|---------------------------------------|--------|-------|-------------------------------------|
| <b>Syntax</b>       | <code>status = MSGQ_alloc(poolId, msg, size);</code>                                                                                                                                                                                                                                                  |                                                |         |                                                |          |       |                                       |        |       |                                     |
| <b>Parameters</b>   | <table><tbody><tr><td>UInt16</td><td>poolId;</td><td>/* allocate the message from this allocator */</td></tr><tr><td>MSGQ_Msg</td><td>*msg;</td><td>/* pointer to the returned message */</td></tr><tr><td>UInt16</td><td>size;</td><td>/* size of the requested message */</td></tr></tbody></table> | UInt16                                         | poolId; | /* allocate the message from this allocator */ | MSGQ_Msg | *msg; | /* pointer to the returned message */ | UInt16 | size; | /* size of the requested message */ |
| UInt16              | poolId;                                                                                                                                                                                                                                                                                               | /* allocate the message from this allocator */ |         |                                                |          |       |                                       |        |       |                                     |
| MSGQ_Msg            | *msg;                                                                                                                                                                                                                                                                                                 | /* pointer to the returned message */          |         |                                                |          |       |                                       |        |       |                                     |
| UInt16              | size;                                                                                                                                                                                                                                                                                                 | /* size of the requested message */            |         |                                                |          |       |                                       |        |       |                                     |
| <b>Return Value</b> | <table><tbody><tr><td>Int</td><td>status;</td><td>/* status */</td></tr></tbody></table>                                                                                                                                                                                                              | Int                                            | status; | /* status */                                   |          |       |                                       |        |       |                                     |
| Int                 | status;                                                                                                                                                                                                                                                                                               | /* status */                                   |         |                                                |          |       |                                       |        |       |                                     |

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

|                    |                                                                                                                   |
|--------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>Description</b> | MSGQ_alloc returns a message from the specified allocator. The size is in minimum addressable data units (MADUs). |
|--------------------|-------------------------------------------------------------------------------------------------------------------|

This function is performed by a writer. This call is non-blocking and can be called from a HWI, SWI or TSK.

All messages must be allocated from an allocator. Once a message is allocated it can be sent. Once a message is received, it must either be freed or re-used.

The poolId must correspond to one of the allocators specified by the allocators field of the POOL\_Config structure specified by the application. (See "Static Configuration" on page 2-327.)

If a message is allocated, SYS\_OK is returned. Otherwise, SYS\_EINVAL is returned if the poolId is invalid, and SYS\_EALLOC is returned if no memory is available to meet the request.

**Constraints and Calling Context**

- ❑ All message definitions must have MSGQ\_MsgHeader as its first field. For example:

```
struct MyMsg {
    MSGQ_MsgHeader header; /* Required field */
    ...                    /* User fields */
}
```

**Example**

```
/* Allocate a message */
status = MSGQ_alloc(STATICPOOLID, (MSGQ_Msg *)&msg,
    sizeof(MyMsg));
if (status != SYS_OK) {
    SYS_abort("Failed to allocate a message");
}
```

**See Also**

MSGQ\_free

**MSGQ\_close***Close a message queue***C Interface**

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = MSGQ_close(msgqQueue);</code>                           |
| <b>Parameters</b>   | <code>MSGQ_Queue     msgqQueue;    /* Message queue to close */</code> |
| <b>Return Value</b> | <code>Int               status;        /* status */</code>             |

**Reentrant**               yes

**Description**               MSGQ\_close closes a message queue. If any messages are in the message queue, they are deleted.

This function is performed by the reader.

Threads that have located (with MSGQ\_locate or MSGQ\_locateAsync) the message queue being closed are not notified about the closure.

If successful, this function returns SYS\_OK.

**Constraints and  
Calling Context**

- ❑ The message queue must have been returned from MSGQ\_open.

**See Also**               MSGQ\_open

**MSGQ\_count***Return the number of messages in a message queue***C Interface****Syntax**

```
status = MSGQ_count(msgqQueue, count);
```

**Parameters**

```
MSGQ_Queue    msgqQueue;    /* Message queue to count */
Uns           *count;        /* Pointer to returned count */
```

**Return Value**

```
Int           status;        /* status */
```

**Reentrant**

```
yes
```

**Description**

This API determines the number of messages in a specific message queue. Only the processor that opened the message queue should call this API to determine the number of messages in the reader's message queue. This API is not thread safe with MSGQ\_get when accessing the same message queue, so the caller of MSGQ\_count must prevent any calls to MSGQ\_get.

If successful, this function returns SYS\_OK.

**Constraints and  
Calling Context**

- ❑ The message queue must have been returned from a MSGQ\_open call.

**Example**

```
status = MSGQ_count(readerMsgQueue, &count);
if (status != SYS_OK) {
    return;
}
LOG_printf(&trace, "There are %d messages.", count);
```

**See Also**

```
MSGQ_open
```



**MSGQ\_free***Free a message***C Interface**

**Syntax**                      `status = MSGQ_free(msg);`

**Parameters**                `MSGQ_Msg msg;        /* Message to be freed */`

**Return Value**              `Int                status;    /* status */`

**Reentrant**                      `yes`

**Description**                `MSGQ_free` frees a message back to the allocator.

If successful, this function returns `SYS_OK`.

This call is non-blocking and can be called from a HWI, SWI or TSK.

**Constraints and Calling Context**      □ The message must have been allocated via `MSGQ_alloc`.

**Example**

```
status = MSGQ_get(readerMsgQueue, (MSGQ_Msg *)msg,
                  SYS_FOREVER);
if (status != SYS_OK) {
    SYS_printf("MSGQ_get call failed.");
}
// process message

MSGQ_free(msg);
```

**See Also**                      `MSGQ_alloc`

**MSGQ\_get***Receive a message from the message queue***C Interface****Syntax**

```
status = MSGQ_get(msgqQueue, msg, timeout);
```

**Parameters**

|            |            |                                       |
|------------|------------|---------------------------------------|
| MSGQ_Queue | msgqQueue; | /* Message queue */                   |
| MSGQ_Msg   | *msg;      | /* Pointer to the returned message */ |
| Uns        | timeout;   | /* Duration to block if no message */ |

**Return Value**

|     |         |              |
|-----|---------|--------------|
| Int | status; | /* status */ |
|-----|---------|--------------|

**Reentrant**

yes

**Description**

MSGQ\_get returns a message sent via MSGQ\_put. The order of retrieval is FIFO.

This function is performed by the reader. Once a message has been received, the reader is responsible for freeing or re-sending the message.

If no messages are present, the pend() function specified in the MSGQ\_Attrs passed to MSGQ\_open for this message queue is called. The pend() function blocks up to the timeout value (SYS\_FOREVER = forever). The timeout units are system clock ticks.

This function is deterministic if timeout is zero. MSGQ\_get can be called from a TSK with any timeout. It can be called from a HWI or SWI if the timeout is zero.

If successful, this function returns SYS\_OK. Otherwise, SYS\_ETIMEOUT is returned if the timeout expires before the message is received.

**Constraints and Calling Context**

- ❑ Only one reader of a message queue is allowed concurrently.
- ❑ The message queue must have been returned from a MSGQ\_open call.

**Example**

```
status = MSGQ_get(readerMsgQueue, (MSGQ_Msg *)&msg, 0);
if (status != SYS_OK) {
    /* No messages to process */
    return;
}
```

**See Also**

MSGQ\_put  
MSGQ\_open

**MSGQ\_getAttrs**

*Returns the attributes of a message queue*

**C Interface**

|                     |                                           |            |                                   |
|---------------------|-------------------------------------------|------------|-----------------------------------|
| <b>Syntax</b>       | status = MSGQ_getAttrs(msgqQueue, attrs); |            |                                   |
| <b>Parameters</b>   | MSGQ_Queue                                | msgqQueue; | /* Message queue */               |
|                     | MSGQ_Attrs                                | *attrs;    | /* Attributes of message queue */ |
| <b>Return Value</b> | Int                                       | status     | /* status */                      |

**Reentrant** yes

**Description** MSGQ\_getAttrs fills in the attrs structure passed to it with the attributes of a local message queue. These attributes are set by MSGQ\_open.

The API returns SYS\_OK unless the message queue is not local (that is, it was opened on another processor). If the message queue is not local, the API returns SYS\_EINVAL and does not change the contents of the passed in attrs structure.

**Example**

```
status = MSGQ_getAttrs (msgqQueue, &attrs);
if (status != SYS_OK) {
    return;
}
notifyHandle = attrs.notifyHandle;
```

- Constraints and Calling Context**
- ❑ The message queue must have been returned from a MSGQ\_open call and must be valid.
  - ❑ This function can be called from a HWI, SWI or TSK.

**See Also** MSGQ\_open

**MSGQ\_getDstQueue**

*Get destination message queue field in a message*

**C Interface**

|                                        |                                                                                                                                                                                                                                                                                                                                 |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | MSGQ_getDstQueue(msg, msgqQueue);                                                                                                                                                                                                                                                                                               |
| <b>Parameters</b>                      | MSGQ_Msg                   msg;                   /* Message */<br>MSGQ_Queue               *msgqQueue;   /* Message queue */                                                                                                                                                                                                   |
| <b>Return Value</b>                    | Void                                                                                                                                                                                                                                                                                                                            |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b>                     | <p>This API allows the application to determine the destination message queue of a message. This API is generally used by transports to determine the final destination of a message. This API can also be used by the application once the message is received.</p> <p>This function can be called from a HWI, SWI or TSK.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ The message must have been sent via MSGQ_put.</li></ul>                                                                                                                                                                                                                                 |

**MSGQ\_getMsgId***Return the message ID from a message***C Interface**

**Syntax** `msgId = MSGQ_getMsgId(msg);`

**Parameters** `MSGQ_Msg msg; /* Message */`

**Return Value** `UInt16 msgId; /* Message ID */`

**Reentrant** `yes`

**Description** MSGQ\_getMsgId returns the message ID from a received message. This message ID is specified via the MSGQ\_setMsgId function.

This function can be called from a HWI, SWI or TSK.

**Example**

```
/* Make sure the message is the one expected */
if (MSGQ_getMsgId((MSGQ_Msg)msg) != MESSAGEID) {
    SYS_abort("Unexpected message");
}
```

**See Also** `MSGQ_setMsgId`

**MSGQ\_getMsgSize***Return the message size from a message***C Interface**

**Syntax**                   size = MSGQ\_getMsgSize(msg);

**Parameters**           MSGQ\_Msg msg;       /\* Message \*/

**Return Value**        Uint16       size;       /\* Message size \*/

**Reentrant**           yes

**Description**        MSGQ\_getMsgSize returns the size of the message buffer out of the received message. The size is in minimum addressable data units (MADUs).

This function can be used to determine if a message can be re-used.

This function can be called from a HWI, SWI or TSK.

**See Also**            MSGQ\_alloc

**MSGQ\_getSrcQueue***Extract the reply destination from a message***C Interface**

|                     |                                            |             |                        |
|---------------------|--------------------------------------------|-------------|------------------------|
| <b>Syntax</b>       | status = MSGQ_getSrcQueue(msg, msgqQueue); |             |                        |
| <b>Parameters</b>   | MSGQ_Msg                                   | msg;        | /* Received message */ |
|                     | MSGQ_Queue                                 | *msgqQueue; | /* Message queue */    |
| <b>Return Value</b> | Int                                        | status;     | /* status */           |
| <b>Reentrant</b>    | yes                                        |             |                        |

**Description**

Many times a receiver of a message wants to reply to the sender of the message (for example, to send an acknowledgement). When a valid msgqQueue is specified in MSGQ\_setSrcQueue, the receiver of the message can extract the message queue via MSGQ\_getSrcQueue.

This is basically the same as a MSGQ\_locate function without knowing the name of the message queue. This function can be used even if the queueName used with MSGQ\_open was NULL or non-unique.

Note: The msgqQueue may not be the sender's message queue handle. The sender is free to use any created message queue handle.

This function can be called from a HWI, SWI or TSK.

If successful, this function returns SYS\_OK.

**Example**

```
/* Get the handle and send the message back. */
status = MSGQ_getSrcQueue((MSGQ_Msg)msg, &replyQueue);
if (status != SYS_OK) {
    /* Free the message and abort */
    MSGQ_free((MSGQ_Msg)msg);
    SYS_abort("Failed to get handle from message");
}
status = MSGQ_put(replyQueue, (MSGQ_Msg)msg);
```

**See Also**

MSGQ\_getAttrs  
MSGQ\_setSrcQueue

**MSGQ\_isLocalQueue**

*Return whether message queue is local or on other processor*

**C Interface**

|                     |                                      |            |                     |
|---------------------|--------------------------------------|------------|---------------------|
| <b>Syntax</b>       | flag = MSGQ_isLocalQueue(msgqQueue); |            |                     |
| <b>Parameters</b>   | MSGQ_Queue                           | msgqQueue; | /* Message queue */ |
| <b>Return Value</b> | Bool                                 | flag;      | /* status */        |

**Reentrant** yes

**Description** This API determines whether the message queue is local (that is, opened on this processor) or remote (that is, opened on a different processor).  
  
If the message queue is local, the flag returned is TRUE. Otherwise, it is FALSE.

**Constraints and Calling Context** ☐ This function can be called from a HWI, SWI or TSK.

**Example**

```
flag = MSGQ_isLocalQueue(readerMsgQueue);
if (flag == TRUE) {
    /* Message queue is local */
    return;
}
```

**See Also** MSGQ\_open



**MSGQ\_locate***Synchronously find a message queue***C Interface**

|                     |                                                                                                                                                                                                                      |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = MSGQ_locate(queueName, msgqQueue, locateAttrs);</code>                                                                                                                                                |
| <b>Parameters</b>   | <p>String            queueName; /* Name of message queue to locate */</p> <p>MSGQ_Queue    *msgqQueue; /* Return located message queue here */</p> <p>MSGQ_LocateAttrs   *locateAttrs;   /* Locate attributes */</p> |
| <b>Return Value</b> | Int                    status;            /* status */                                                                                                                                                               |
| <b>Reentrant</b>    | yes                                                                                                                                                                                                                  |

**Description**

The MSGQ\_locate function is used to locate an opened message queue. This function is synchronous (that is, it can block if timeout is non-zero).

This function is performed by a writer. The reader must have already called MSGQ\_open for this queueName.

MSGQ\_locate firsts searches the local message queues for a name match. If a match is found, that message queue is returned. If no match is found, the transports are queried one at a time. If a transport locates the queueName, that message queue is returned. If the transport does not locate the message queue, the next transport is queried. If no transport can locate the message queue, an error is returned.

In a multiple-processor environment, transports can block when they are queried if you call MSGQ\_locate. The timeout in the MSGQ\_LocateAttrs structure specifies the maximum time each transport can block. The default is SYS\_FOREVER (that is, each transport can block forever). Remember that if you specify 1000 clock ticks as the timeout, the total blocking time could be 1000 \* number of transports.

Note that timeout is not a fixed amount of time to wait. It is the maximum time each transport waits for a positive or negative response. For example, suppose your timeout is 1000, but the response (found or not found) comes back in 600 ticks. The transport returns the response then; it does not wait for another 400 ticks to recheck for a change.

If you do not want to allow blocking, call MSGQ\_locateAsync instead of MSGQ\_locate.

The locateAttrs parameter is of type MSGQ\_LocateAttrs. This type has the following structure:

```
typedef struct MSGQ_LocateAttrs {
    Uns          timeout;
} MSGQ_LocateAttrs;
```

The timeout is the maximum time a transport can block on a synchronous locate in system clock ticks. The default attributes are as follows:

```
MSGQ_LocateAttrs  MSGQ_LOCATEATTRS = {SYS_FOREVER};
```

If successful, this function returns SYS\_OK. Otherwise, it returns SYS\_ENOTFOUND to indicate that it could not locate the specified message queue.

### Constraints and Calling Context

- ❑ Cannot be called from main().
- ❑ Cannot be called in a SWI or HWI context.

### Example

```
status = MSGQ_locate("reader", &readerMsgQueue, NULL);
if (status != SYS_OK) {
    SYS_abort("Failed to locate reader message queue");
}
```

### See Also

MSGQ\_locateAsync  
MSGQ\_open

**MSGQ\_locateAsync***Asynchronously find a message queue***C Interface**

**Syntax** `status = MSGQ_locateAsync(queueName, replyQueue, locateAsyncAttrs);`

**Parameters** `String queueName; /* Name of message queue to locate */`  
`MSGQ_Queue replyQueue; /* Msgq to send locate message */`  
`MSGQ_LocateAsyncAttrs *locateAsyncAttrs; /* Locate attributes */`

**Return Value** `Int status; /* status */`

**Reentrant** `yes`

**Description** MSGQ\_locateAsync firsts searches the local message queues for a name match. If one is found, an asynchronous locate message is sent to the specified message queue (in the replyQueue parameter). If it is not, all transports are asked to start an asynchronous locate search. After all transports have been asked to start the search, the API returns.

If a transport locates the message queue, an asynchronous locate message is sent to the specified replyQueue. If no transport can locate the message queue, no message is sent.

This function is performed by a writer. The reader must have already called MSGQ\_open for this queueName. An asynchronous locate can be performed from a SWI or TSK. It cannot be performed in main().

The message ID for an asynchronous locate message is:

```
/* Asynchronous locate message ID */
#define MSGQ_ASYNCLOCATEMSGID 0xFF00
```

The MSGQ\_LocateAsyncAttrs structure has the following fields:

```
typedef struct MSGQ_LocateAsyncAttrs {
    Uint16 poolId;
    Arg arg;
} MSGQ_LocateAttrs;
```

The default attributes are as follows:

```
MSGQ_LocateAsyncAttrs MSGQ_LOCATEASYNCATTRS = {0, 0};
```

The locate message is allocated from the allocator specified by the locateAsyncAttrs->poolId field.

The locateAsyncAttrs->arg value is included in the asynchronous locate message. This field allows you to correlate requests with the responses.

Once the application receives an asynchronous locate message, it is responsible for freeing the message. The asynchronous locate message received by the replyQueue has the following structure:

```
typedef struct MSGQ_AsyncLocateMsg {
    MSGQ_MsgHeader  header;
    MSGQ_Queue      msgqQueue;
    Arg             arg;
} MSGQ_AsyncLocateMsg;
```

| Field     | Type           | Description                                                       |
|-----------|----------------|-------------------------------------------------------------------|
| header    | MSGQ_MsgHeader | Required field for every message.                                 |
| msgqQueue | MSGQ_Queue     | Located message queue handle.                                     |
| Arg       | Arg            | Value specified in MSGQ_LocateAttrs for this asynchronous locate. |

This function returns SYS\_OK to indicated that an asynchronous locate was started. This status does not indicate whether or not the locate will be successful. The SYS\_EALLOC status is returned if the message could not be allocated.

Constraints and Calling Context

- ❑ The allocator must be able to allocate an asynchronous locate message.
- ❑ Cannot be called in the context of main().

Example

The following example shows an asynchronous locate performed in a task. Time spent blocking is dictated by the timeout specified in the MSGQ\_get call. (Error handling statements were omitted for brevity.)

```
status = MSGQ_open("myMsgQueue", &myQueue, &msgqAttrs);

locateAsyncAttrs          = MSGQ_LOCATEATTRS;
locateAsyncAttrs.poolId   = STATICPOOLID;

MSGQ_locateAsync("msgQ1", myQueue, &locateAsyncAttrs);
status = MSGQ_get(myQueue, &msg, SYS_FOREVER);
if (MSGQ_getMsgId((MSGQ_Msg)msg) ==
    MSGQ_ASYNCLOCATEMSGID) {
    readerQueue = msg->msgqQueue;
}
MSGQ_free((MSGQ_Msg)msg);
```

See Also

MSGQ\_locate  
MSGQ\_free  
MSGQ\_open

MSGQ\_open

Open a message queue

C Interface

|              |                                                                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | status = MSGQ_open(queueName, msgqQueue, attrs);                                                                                                                                       |
| Parameters   | String queueName; /* Unique name of the message queue */<br>MSGQ_Queue *msgqQueue; /* Pointer to returned message queue */<br>MSGQ_Attrs *attrs; /* Attributes of the message queue */ |
| Return Value | Int status; /* status */                                                                                                                                                               |
| Reentrant    | yes                                                                                                                                                                                    |

**Description** MSGQ\_open is the function to open a message queue. This function selects and returns a message queue from the array provided in the static configuration (that is, MSGQ\_config->msgqQueues).

This function is on the processor where the reader resides. The reader then uses this message queue to receive messages.

If successful, this function returns SYS\_OK. Otherwise, it returns SYS\_ENOTFOUND to indicate that no empty spot was available in the message queue array.

If the application will use MSGQ\_locate or MSGQ\_locateAsync to find this message queue, the queueName must be unique. If the application will never need to use the locate APIs, the queueName may be NULL or a non-unique name.

Instead of using a fixed notification mechanism, such as SEM\_pend and SEM\_post, the MSGQ notification mechanism is supplied in the attrs parameter, which is of type MSGQ\_Attrs. If attrs is NULL, the new message queue is assigned a default set of attributes. The structure for MSGQ\_Attrs is as follows:

```
typedef struct MSGQ_Attrs {
    Ptr notifyHandle;
    MSGQ_Pend pend;
    MSGQ_Post post;
} MSGQ_Attrs;
```

The MSGQ\_Attrs fields are as follows:

| Field        | Type | Description                                       |
|--------------|------|---------------------------------------------------|
| notifyHandle | Ptr  | Handle to use in the pend() and post() functions. |

|      |           |                                                     |
|------|-----------|-----------------------------------------------------|
| Pend | MSGQ_Pend | Function pointer to a user-specified pend function. |
| Post | MSGQ_Post | Function pointer to a user-specified post function. |

The default attributes are:

```
MSGQ_Attrs MSGQ_ATTRS = {  
    NULL, /* notifyHandle */  
    (MSGQ_Pend)SYS_zero, /* NOP pend */  
    FXN_F_nop /* NOP post */  
};
```

The following typedefs are provided by the MSGQ module to allow easier casting of the pend and post functions:

```
typedef Bool (*MSGQ_Pend)(Ptr notifyHandle, Uns timeout);  
typedef Void (*MSGQ_Post)(Ptr notifyHandle);
```

The post() function you specify is always called within MSGQ\_put when a writer sends a message.

A reader calls MSGQ\_get to receive a message. If there is a message, it returns that message, and the pend() function is not called. The pend() function is only called if there are no messages to receive.

The pend() and post() functions must act in a binary manner. For instance, SEM\_pend and SEM\_post treat the semaphore as a counting semaphore instead of binary. So SEM\_pend and SEM\_post are an invalid pend/post pair. The following example, in which the reader calls MSGQ\_get with a timeout of SYS\_FOREVER, shows why:

- 1) A writer sends 10 messages, making the count 10 in the semaphore.
- 2) The reader then calls MSGQ\_get 10 times. Each call returns a message without calling the pend() function.
- 3) The reader then calls MSGQ\_get again. Since there are no messages, the pend() function is called. Since the semaphore count was 10, SEM\_pend returns TRUE immediately from the pend(). MSGQ would check for messages and there would still be none, so pend() would be called again. This would repeat 9 more times until the count was zero.

If the pend() function were binary (for example, a binary semaphore), the pend() function would be called at most two times in step 3.

So instead of using SEM\_pend and SEM\_post for synchronous (blocking) opens, you should use SEM\_pendBinary and SEM\_postBinary.

The following notification attributes could be used if the reader is a SWI function (which cannot block):

```
MSGQ_Attrs attrs    = MSGQ_ATTRS; // default attributes
// leave attrs.pend as a NOP
attrs.notifyHandle = (Ptr)swiHandle;
attrs.post         = (MSGQ_Pend)SWI_post;
```

The following notification attributes could be used if the reader is a TSK function (which can block):

```
MSGQ_Attrs attrs    = MSGQ_ATTRS; // default attributes
attrs.notifyHandle = (Ptr)semHandle;
attrs.pend         = (MSGQ_Pend)SEM_pendBinary;
attrs.post         = (MSGQ_Post)SEM_postBinary;
```

## Constraints and Calling Context

- ❑ The message queue returned is to be used by the caller of MSGQ\_get. It should not be used by writers to that message queue (that is, callers of MSGQ\_put). Writers should use the message queue returned by MSGQ\_locate, MSGQ\_locateAsync, or MSGQ\_getSrcQueue.
- ❑ If a post() function is specified, the function must be non-blocking.
- ❑ If a pend() function is specified, the function must be non-blocking when timeout is zero.
- ❑ Each message queue must have a unique name if the application will use MSGQ\_locate or MSGQ\_locateAsync.
- ❑ The queueName must be persistent. The MSGQ module references this name internally; that is, it does not make a copy of the name.

## Example

```
/* Open the reader message queue.
 * Using semaphores as notification mechanism */
msgqAttrs    = MSGQ_ATTRS;
msgqAttrs.notifyHandle = (Ptr)readerSemHandle;
msgqAttrs.pend         = (MSGQ_Pend)SEM_pendBinary;
msgqAttrs.post         = (MSGQ_Post)SEM_postBinary;
status = MSGQ_open("reader", &readerMsgQueue,
                  &msgqAttrs);
if (status != SYS_OK) {
    SYS_abort("Failed to open the reader message queue");
}
```

## See Also

MSGQ\_close  
 MSGQ\_locate  
 MSGQ\_locateAsync  
 SEM\_pendBinary  
 SEM\_postBinary

MSGQ\_put

Place a message on a message queue

C Interface

|              |                                                             |            |                                 |
|--------------|-------------------------------------------------------------|------------|---------------------------------|
| Syntax       | status = MSGQ_put(msgqQueue, msg);                          |            |                                 |
| Parameters   | MSGQ_Queue                                                  | msgqQueue; | /* Destination message queue */ |
|              | MSGQ_Msg                                                    | msg;       | /* Message */                   |
| Return Value | Int                                                         | status;    | /* status */                    |
| Reentrant    | yes                                                         |            |                                 |
| Description  | MSGQ_put places a message into the specified message queue. |            |                                 |

This function is performed by a writer. This function is non-blocking, and can be called from a HWI, SWI or TSK.

The post() function for the destination message queue is called as part of the MSGQ\_put. The post() function is specified MSGQ\_open call in the MSGQ\_Attrs parameter.

If successful, this function returns SYS\_OK. Otherwise, it may return an error code returned by the transport.

There are several features available when sending a message.

- ❑ A msgId passed to MSGQ\_setMsgId can be used to indicate the type of message it is. Such a type is completely application-specific, except for IDs defined for MSGQ\_setMsgId. The reader of a message can use MSGQ\_getMsgId to get the ID from the message.
- ❑ The source message queue parameter to MSGQ\_setSrcQueue allows the sender of the message to specify a source message queue. The receiver of the message can use MSGQ\_getSrcQueue to extract the embedded message queue from the message. A client/server application might use this mechanism because it allows the server to reply to a message without first locating the sender. For example, each client would have its own message queue that it specifies as the source message queue when it sends a message to the server. The server can use MSGQ\_getSrcQueue to get the message queue to reply back to.

If MSGQ\_put returns an error, the user still owns the message and is responsible for freeing the message (or re-sending it).

Constraints and Calling Context

- ❑ The msgqQueue must have been returned from MSGQ\_locate, MSGQ\_locateAsync or MSGQ\_getSrcQueue (or MSGQ\_open if the reader of the message queue wants to send themselves a message).



- ❑ If MSGQ\_put does not return SYS\_OK, the message is still owned by the caller and must either be freed or re-used.

**Example**

```
/* Send the message back. */
status = MSGQ_put(replyMsgQueue, (MSGQ_Msg)msg);
if (status != SYS_OK) {
    /* Need to free the message */
    MSGQ_free((MSGQ_Msg)msg);
    SYS_abort("Failed to send the message");
}
```

**See Also**

MSGQ\_get  
MSGQ\_open  
MSGQ\_setMsgId  
MSGQ\_getMsgId  
MSGQ\_setSrcQueue  
MSGQ\_getSrcQueue

**MSGQ\_release**

*Release a located message queue*

**C Interface**

|                                        |                                                                                                                                                                                                                                                                                                                 |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | status = MSGQ_release(msgqQueue);                                                                                                                                                                                                                                                                               |
| <b>Parameters</b>                      | MSGQ_Queue    msgqQueue;    /* Message queue to release */                                                                                                                                                                                                                                                      |
| <b>Return Value</b>                    | Int                    status;                    /* status */                                                                                                                                                                                                                                                  |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                                                                                                             |
| <b>Description</b>                     | <p>This function releases a located message queue. That is, it releases a message queue returned from MSGQ_locate or MSGQ_locateAsync.</p> <p>This function is performed by a writer.</p> <p>If successful, this function returns SYS_OK. Otherwise, it may return an error code returned by the transport.</p> |
| <b>Constraints and Calling Context</b> | <div>❑ The handle must have been returned from MSGQ_locate or MSGQ_locateAsync.</div>                                                                                                                                                                                                                           |
| <b>See Also</b>                        | MSGQ_locate<br>MSGQ_locateAsync                                                                                                                                                                                                                                                                                 |

## **MSGQ\_setErrorHandler** *Set up handling of internal MSGQ errors*

### **C Interface**

|                     |                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = MSGQ_setErrorHandler(errorQueue, poolId);</code>                                                                                     |
| <b>Parameters</b>   | <code>MSGQ_Queue errorQueue; /* Message queue to receive errors */</code><br><code>UInt16 poolId; /* Allocator to allocate error messages */</code> |
| <b>Return Value</b> | <code>Int status; /* status */</code>                                                                                                               |
| <b>Reentrant</b>    | yes                                                                                                                                                 |

### **Description**

Asynchronous errors that need to be communicated to the application may occur in a transport. If an application calls `MSGQ_setErrorHandler`, all asynchronous errors are then sent to the message queue specified.

The specified message queue receives asynchronous error messages (if they occur) via `MSGQ_get`.

`poolId` specifies the allocator the transport should use to allocate error messages. If the transports cannot allocate a message, no action is performed.

If this function is not called or if `errorHandler` is set to `MSGQ_INVALIDMSGQ`, no error messages will be allocated and sent.

This function can be called multiple times with only the last handler being active.

If successful, this function returns `SYS_OK`.

The message ID for an asynchronous error message is:

```
/* Asynchronous error message ID */
#define MSGQ_ASYNCERRORMSGID 0xFF01
```

The following is the structure for an asynchronous error message:

```
typedef struct MSGQ_AsyncErrorMsg {
    MSGQ_MsgHeader header;
    MSGQ_MqtError errorType;
    UInt16 mqtId;
    UInt16 parameter;
} MSGQ_AsyncErrorMsg;
```

The following table describes the fields in the MSGQ\_AsyncErrorMsg structure:

| Field     | Type           | Description                                     |
|-----------|----------------|-------------------------------------------------|
| header    | MSGQ_MsgHeader | Required field for every message                |
| errorType | MSGQ_MqtError  | Error ID                                        |
| mqtId     | UInt16         | ID of the transport that sent the error message |
| parameter | UInt16         | Error-specific field                            |

The following table lists the valid errorType values and the meanings of their arg fields:

| errorType             | mqtId                                              | parameter                                                                                                                                                                                                                                                                                                    |
|-----------------------|----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MSGQ_MQTTEROREXIT     | ID of the transport that is exiting.               | Not used.                                                                                                                                                                                                                                                                                                    |
| MSGQ_MQTFAILEDPUT     | ID of the transport that failed to send a message. | ID of destination queue. The parameter is 16 bits, so only the lower 16 bits of the msgQueue is logged. The top 16 bits of the msgQueue contain the destination processor ID, which is also the mqtId. You can OR the mqtId shifted over by 16 bits with the parameter to get the full destination msgQueue. |
| MSGQ_MQTERRORINTERNAL | Generic internal error.                            | Transport defined.                                                                                                                                                                                                                                                                                           |
| MSGQ_MQTERRORPHYSICAL | Problem with the physical link.                    | Transport defined.                                                                                                                                                                                                                                                                                           |
| MSGQ_MQTERRORALLOC    | Transport could not allocate memory.               | Size of the requested memory.                                                                                                                                                                                                                                                                                |

MSGQ\_open  
MSGQ\_get

**MSGQ\_setMsgId**

*Set the message ID in a message*

**C Interface**

**Syntax** MSGQ\_setMsgId(msg, msgId);

**Parameters** MSGQ\_MSG msg; /\* Message \*/  
                  Uint16 msgId; /\* Message id \*/

**Return Value** Void

**Reentrant** yes

**Description** Inside each message is a message id field. This API sets this field. The value of msgId is application-specific. MSGQ\_getMsgId can be used to extract this field from a message.

When a message is allocated, the value of this field is MSGQ\_INVALIDMSGID. When MSGQ\_setMsgId is called, it updates the field accordingly. This API can be called multiple times on a message.

If a message is sent to another processor, the message Id field is converted by the transports accordingly (for example, endian conversion is performed).

The message IDs used when sending messages are application-specific. They can have any value except values in the following ranges:

- ❑ Reserved for the MSGQ module messages: 0xFF00 - 0xFF7F
- ❑ Reserved for internal transport usage: 0xFF80 - 0xFFFE
- ❑ Used to signify an invalid message ID: 0xFFFF

The following table lists the message IDs currently used by the MSGQ module.

| Constant Defined in msgq.h | Value  | Description                                      |
|----------------------------|--------|--------------------------------------------------|
| MSGQ_ASYNCLOCATEMSGID      | 0xFF00 | Used to denote an asynchronous locate message.   |
| MSGQ_ASYNCERRORMSGID       | 0xFF01 | Used to denote an asynchronous transport error.  |
| MSGQ_INVALIDMSGID          | 0xFFFF | Used as initial value when message is allocated. |

**Constraints and Calling Context**      ❑ Message must have been allocated originally from MSGQ\_alloc.

**Example**

```
/* Fill in the message */
msg->sequenceNumber = 0;
MSGQ_setMsgId((MSGQ_Msg)msg, MESSAGEID);

/* Send the message */
status = MSGQ_put(readerMsgQueue, (MSGQ_Msg)msg);
    if (status != SYS_OK) {
        SYS_abort("Failed to send the message");
    }
```

**See Also**

MSGQ\_getMsgId  
MSGQ\_setErrorHandler

**MSGQ\_setSrcQueue***Set the reply destination in a message***C Interface**

|                     |                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | MSGQ_setSrcQueue(msg, msgQueue);                                                           |
| <b>Parameters</b>   | MSGQ_MSG      msg;            /* Message */<br>MSGQ_Queue    msgQueue; /* Message queue */ |
| <b>Return Value</b> | Void                                                                                       |
| <b>Reentrant</b>    | yes                                                                                        |

**Description**

This API allows the sender to specify a message queue that the receiver of the message can reply back to (via MSGQ\_getSrcQueue). The msgQueue must have been returned by MSGQ\_open.

Inside each message is a source message queue field. When a message is allocated, the value of this field is MSGQ\_INVALIDMSGQ. When this API is called, it updates the field accordingly. This API can be called multiple times on a message.

If a message is sent to another processor, the source message queue field is managed by the transports accordingly.

**Constraints and Calling Context**

- ❑ Message must have been allocated originally from MSGQ\_alloc.
- ❑ msgQueue must have been returned from MSGQ\_open.

**Example**

```
/* Fill in the message */
msg->sequenceNumber = 0;
MSGQ_setSrcQueue((MSGQ_Msg)msg, writerMsgQueue);

/* Send the message */
status = MSGQ_put(readerMsgQueue, (MSGQ_Msg)msg);
if (status != SYS_OK) {
    SYS_abort("Failed to send the message");
}
```

**See Also**

MSGQ\_getSrcQueue

## 2.20 PIP Module

---

**Important Note:** The PIP module is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

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.



**Configuration Properties**

- ❑ **Uns writerSize.** Number of words available in the frame allocated with PIP\_alloc.
- ❑ **Uns writerNumFrames.** Number of frames available to be written to.

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PIP Manager Properties and PIP Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

| Name      | Type      | Default           |
|-----------|-----------|-------------------|
| OBJMEMSEG | Reference | prog.get("IDRAM") |

**Instance Configuration Parameters**

| Name             | Type       | Default (Enum Options)      |
|------------------|------------|-----------------------------|
| comment          | String     | "<add comments here>"       |
| bufSeg           | Reference  | prog.get("IDRAM")           |
| bufAlign         | Int16      | 1                           |
| frameSize        | Int16      | 8                           |
| numFrames        | Int16      | 2                           |
| monitor          | EnumString | "reader" ("writer", "none") |
| notifyWriterFxn  | Extern     | prog.extern("FXN_F_nop")    |
| notifyWriterArg0 | Arg        | 0                           |
| notifyWriterArg1 | Arg        | 0                           |
| notifyReaderFxn  | Extern     | prog.extern("FXN_F_nop")    |
| notifyReaderArg0 | Arg        | 0                           |
| notifyReaderArg1 | Arg        | 0                           |

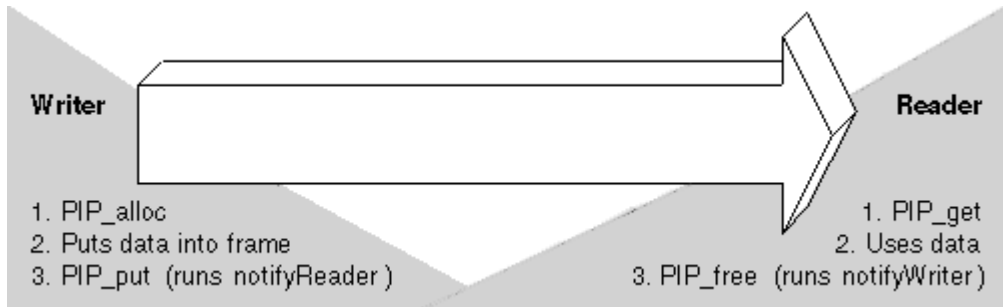
**Description**

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 2-4. 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 2-4. 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 SWI 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.

**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, such as SWI\_andnHook.

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 a SWI, but pipes can also be used to transfer data between two application threads.

The following global property can be set for the PIP module in the PIP Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.PIP.OBJMEMSEG = prog.get("myMEM");`

**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).

To create a PIP object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myPip = bios.PIP.create("myPip");
```

The following properties can be set for a PIP object in the PIP Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this PIP object.  
 Tconf Name: comment Type: String  
 Example: `myPip.comment = "my PIP";`
- ❑ **bufseg.** The memory segment that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes).  
 Tconf Name: bufSeg Type: Reference  
 Example: `myPip.bufSeg = prog.get("myMEM");`
- ❑ **bufalign.** The alignment (in words) of the buffer allocated within the specified memory segment.  
 Tconf Name: bufAlign Type: Int16  
 Example: `myPip.bufAlign = 1;`
- ❑ **framesize.** The length of each frame (in words)  
 Tconf Name: frameSize Type: Int16  
 Example: `myPip.frameSize = 8;`
- ❑ **numframes.** The number of frames  
 Tconf Name: numFrames Type: Int16  
 Example: `myPip.numFrames = 2;`
- ❑ **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.  
 Tconf Name: monitor Type: EnumString  
 Options: "reader", "writer", "none"  
 Example: `myPip.monitor = "reader";`
- ❑ **notifyWriter.** The function to execute when a frame of free space is available. This function should notify (for example, by calling SWI\_andnHook) 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.

Tconf Name: notifyWriterFxn Type: Extern

Example: `myPip.notifyWriterFxn =  
          prog.extern("writerFxn");`

- ❑ **nwarg0, nwarg1.** Two Arg type arguments for the notifyWriter function.

Tconf Name: notifyWriterArg0 Type: Arg

Tconf Name: notifyWriterArg1 Type: Arg

Example: `myPip.notifyWriterArg0 = 0;`

- ❑ **notifyReader.** The function to execute when a frame of data is available. 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\_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.

Tconf Name: notifyReaderFxn Type: Extern

Example: `myPip.notifyReaderFxn =  
          prog.extern("readerFxn");`

- ❑ **nrarg0, nrarg1.** Two Arg type arguments for the notifyReader function.

Tconf Name: notifyReaderArg0 Type: Arg

Tconf Name: notifyReaderArg1 Type: Arg

Example: `myPip.notifyReaderArg0 = 0;`

PIP\_alloc

Allocate an empty frame from a pipe

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax                          | PIP_alloc(pipe);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Parameters                      | PIP_Handle pipe; /* pipe object handle */                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Return Value                    | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Reentrant                       | no                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description                     | <p>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.</p> <p>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_andnHook) 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.</p> |
| Constraints and Calling Context | <ul style="list-style-type: none"><li>❑ 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).</li><li>❑ PIP_alloc can only be called one time before calling PIP_put. You cannot operate on two frames from the same pipe simultaneously.</li></ul>                                                                                                                                                                                                                                                                    |

**Note:**

Registers used by notifyWriter functions might also be modified.

**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 2-172, 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*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface**

|                     |                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | PIP_free(pipe);                                                                                                             |
| <b>Parameters</b>   | PIP_Handle pipe; /* pipe object handle */                                                                                   |
| <b>Return Value</b> | Void                                                                                                                        |
| <b>Reentrant</b>    | no                                                                                                                          |
| <b>Description</b>  | 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\_andnHook) 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.

|                                        |                                                                                                                                                                                              |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Constraints and Calling Context</b> | <div><input type="checkbox"/> When called within an HWI, the code sequence calling PIP_free must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.</div> |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

---

**Note:**  
Registers used by notifyWriter functions might also be modified.

---

|                |                                                                                                                                 |
|----------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>Example</b> | See the example for PIP_alloc, page 2–312. The example for HST_getpipe, page 2–172, also uses a pipe with host channel objects. |
|----------------|---------------------------------------------------------------------------------------------------------------------------------|

|                 |                                                |
|-----------------|------------------------------------------------|
| <b>See Also</b> | PIP_alloc<br>PIP_get<br>PIP_put<br>HST_getpipe |
|-----------------|------------------------------------------------|



**PIP\_get***Get a full frame from the pipe*


---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

PIP\_get(pipe);

**Parameters**

PIP\_Handle pipe; /\* pipe object handle \*/

**Return Value**

Void

**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\_andnHook) 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.

---

**Note:** Registers used by notifyReader functions might also be modified.

---

**Example**

See the example for PIP\_alloc, page 2–312. The example for HST\_getpipe, page 2–172, 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*


---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

```
readerAddr = PIP_getReaderAddr(pipe);
```

**Parameters**

```
PIP_Handle pipe;    /* pipe object handle */
```

**Return Value**

```
Ptr          readerAddr
```

**Reentrant**

```
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*

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

### 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 \*/

#### 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 2–316.

**PIP\_getReaderSize***Get the number of words of data in a pipe frame*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**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 */
```

**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 2–316.

**PIP\_getWriterAddr***Get the value of the writerAddr pointer of the pipe*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

```
writerAddr = PIP_getWriterAddr(pipe);
```

**Parameters**

```
PIP_Handle pipe;    /* pipe object handle */
```

**Return Value**

```
Ptr          writerAddr;
```

**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 2–316.

**PIP\_getWriterNumFrames**

*Get number of pipe frames available to be written to*

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

**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 */                                                                                                                                                                                                                |
| <b>Reentrant</b>    | yes                                                                                                                                                                                                                                                                |
| <b>Description</b>  | <p>PIP_getWriterNumFrames is a C function that returns the value of the writerNumFrames element of a pipe object.</p> <p>Before a function attempts to write to a pipe, it should call PIP_getWriterNumFrames to ensure at least one empty frame is available.</p> |
| <b>Example</b>      | See the example for PIP_getReaderAddr, page 2–316.                                                                                                                                                                                                                 |

**PIP\_getWriterSize***Get the number of words that can be written to a pipe frame*


---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

```
num = PIP_getWriterSize(pipe);
```

**Parameters**

```
PIP_Handle pipe;    /* pipe object handle*/
```

**Return Value**

```
Uns      num;      /* num of words to be written in empty frame */
```

**Reentrant**

```
yes
```

**Description**

PIP\_getWriterSize is a C function that returns the value of the writerSize element of a pipe object.

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.

**Example**

```
if (PIP_getWriterNumFrames(rxPipe) > 0) {
    PIP_alloc(rxPipe);
    DSS_rxPtr = PIP_getWriterAddr(rxPipe);
    DSS_rxCnt = PIP_getWriterSize(rxPipe);
}
```

**PIP\_peek***Get pipe frame size and address without actually claiming pipe frame*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

```
framesize = PIP_peek(pipe, addr, rw);
```

**Parameters**

```
PIP_Handle  pipe;    /* pipe object handle */
Ptr         *addr;   /* address of variable with frame address */
Uns        rw;      /* flag to indicate the reader or writer side */
```

**Return Value**

```
Int         framesize; /* the frame size */
```

**Description**

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.

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.

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.

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.

**See Also**

```
PIP_alloc
PIP_free
PIP_get
PIP_put
PIP_reset
```



PIP\_put

Put a full frame into the pipe

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

|              |                                           |
|--------------|-------------------------------------------|
| Syntax       | PIP_put(pipe);                            |
| Parameters   | PIP_Handle pipe; /* pipe object handle */ |
| Return Value | Void                                      |
| Reentrant    | no                                        |

Description

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.

**Note:** Registers used by notifyReader functions might also be modified.

Constraints and Calling Context

- ❑ When called within an HWI, 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 2–312. The example for HST\_getpipe, page 2–172, 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*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

PIP\_reset(pipe);

**Parameters**

PIP\_Handle pipe;     /\* pipe object handle \*/

**Return Value**

Void

**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*

---

**Important Note:** This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

---

**C Interface****Syntax**

```
PIP_setWriterSize(pipe, size);
```

**Parameters**

```
PIP_Handle pipe;    /* pipe object handle */  
Uns        size;    /* size to be set */
```

**Return Value**

```
Void
```

**Reentrant**

```
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 2–316.

## 2.21 POOL Module

Functions

The POOL module describes the interface that allocators must provide.

None; this module describes an interface to be implemented by allocators

Constants, Types, and Structures

```
POOL_Config POOL_config;

typedef struct POOL_Config {
    POOL_Obj *allocators;    /* Array of allocators */
    Uint16    numAllocators; /* Num of allocators */
} POOL_Config;

typedef struct POOL_Obj {
    POOL_Init  initFxn; /* Allocator init function */
    POOL_Fxns *fxns;    /* Interface functions */
    Ptr        params;  /* Setup parameters */
    Ptr        object;   /* Allocator's object */
} POOL_Obj, *POOL_Handle;
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the POOL Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

### Module Configuration Parameters

| Name       | Type | Default (Enum Options) |
|------------|------|------------------------|
| ENABLEPOOL | Bool | false                  |

Description

The POOL module describes standard interface functions that allocators must provide. The allocator interface functions are called internally by the MSGQ module and not by user applications. A simple static allocator, called STATICPOOL, is provided with DSP/BIOS. Other allocators can be implemented by following the standard interface.

**Note:** This document does not discuss how to write an allocator. Information about designing allocators will be provided in a future document.

All messages sent via the MSGQ module must be allocated by an allocator. The allocator determines where and how the memory for the message is allocated.

An allocator is an instance of an implementation of the allocator interface. An application may instantiate one or more instances of an allocator.

An application can use multiple allocators. The purpose of having multiple allocators is to allow an application to regulate its message usage. For example, an application can allocate critical messages from one pool of fast on-chip memory and non-critical messages from another pool of slower external memory.

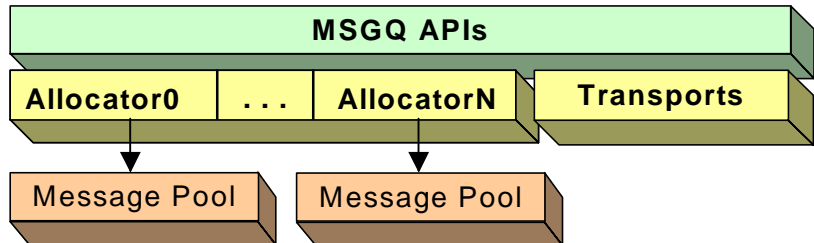


Figure 2-5. Allocators and Message Pools

### Static Configuration

In order to use an allocator and the POOL module, you must statically configure the following:

- ❑ ENABLEPOOL property of the POOL module using Tconf (see “POOL Manager Properties” on page 2-330)
- ❑ POOL\_config variable in application code (see below)

An application must provide a filled in POOL\_config variable if it uses one or more allocators.

```
POOL_Config POOL_config;
```

Where the POOL\_Config structure has the following structure:

```
typedef struct POOL_Config {
    POOL_Obj *allocators; /* Array of allocators */
    Uint16    numAllocators; /* Num of allocators */
} POOL_Config;
```

The fields in this structure are as follows:

| Field         | Type     | Description                                  |
|---------------|----------|----------------------------------------------|
| allocators    | POOL_Obj | Array of allocator objects                   |
| numAllocators | Uint16   | Number of allocators in the allocator array. |

If the POOL module is enabled via Tconf and the application does not provide the POOL\_config variable, the application cannot be linked successfully.

The following is the POOL\_Obj structure:

```
typedef struct POOL_Obj {
    POOL_Init  initFxn; /* Allocator init function */
    POOL_Fxns *fxns;    /* Interface functions */
    Ptr        params;  /* Setup parameters */
    Ptr        object;  /* Allocator's object */
} POOL_Obj, *POOL_Handle;
```

The fields in the POOL\_Obj structure are as follows:

| Field   | Type        | Description                                                                                                                                                                |
|---------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| initFxn | POOL_Init   | Initialization function for this allocator. This function will be called during DSP/BIOS initialization. More explicitly it is called before main().                       |
| fxns    | POOL_Fxns * | Pointer to the allocator's interface functions.                                                                                                                            |
| params  | Ptr         | Pointer to the allocator's parameters. This field is allocator-specific. Please see the documentation provided with your allocator for a description of this field.        |
| object  | Ptr         | State information needed by the allocator. This field is initialized and managed by the allocator. See the allocator documentation to determine how to specify this field. |

One allocator implementation (STATICPOOL) is shipped with DSP/BIOS. Additional allocator implementations can be created by application writers.

## STATICPOOL Allocator

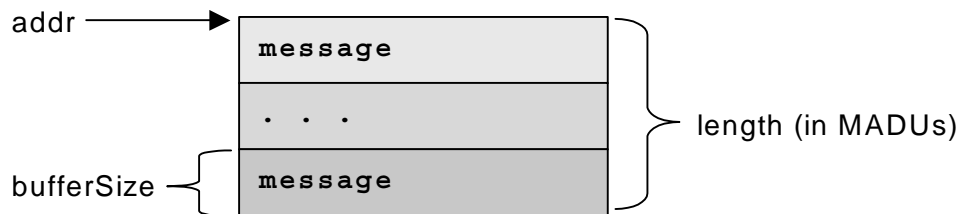
The STATICPOOL allocator takes a user-specified buffer and allocates fixed-size messages from the buffer. The following are its configuration parameters:

```
typedef struct STATICPOOL_Params {
    Ptr        addr;
    size_t     length;
    size_t     bufferSize;
} STATICPOOL_Params;
```

The following table describes the fields in this structure:

| Field      | Type   | Description                                                                                                                                                                                                                                                                                                                                      |
|------------|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| addr       | Ptr    | User supplied block of memory for allocating messages from. The address will be aligned on an 8 MADU boundary for correct structure alignment on all ISAs. If there is a chance the buffer is not aligned, allow at least 7 extra MADUs of space to allow room for the alignment. You can use the DATA_ALIGN pragma to force alignment yourself. |
| length     | size_t | Size of the block of memory pointed to by addr.                                                                                                                                                                                                                                                                                                  |
| bufferSize | size_t | Size of the buffers in the block of memory. The bufferSize must be a multiple of 8 to allow correct structure alignment.                                                                                                                                                                                                                         |

The following figure shows how the fields in STATICPOOL\_Params define the layout of the buffer:



*Figure 2-6. Buffer Layout as Defined by STATICPOOL\_Params*

Since the STATICPOOL buffer is generally used in static systems, the application must provide the memory for the STATICPOOL\_Obj. So the object field of the POOL\_Obj must be set to STATICPOOL\_Obj instead of NULL.

The following is an example of an application that has two allocators (two instances of the STATICPOOL implementation).

```
#define NUMMSGs 8 /* Number of msgs per allocator */

/* Size of messages in the two allocators. Must be a
 * multiple of 8 as required by static allocator. */
#define MSGSIZE0 64
#define MSGSIZE1 128

enum { /* Allocator ID and number of allocators */
    MQASTATICID0 = 0,
    MQASTATICID1,
    NUMALLOCATORS
};
```

```
#pragma DATA_ALIGN(staticBuf0, 8) /* As required */
#pragma DATA_ALIGN(staticBuf1, 8) /* As required */
static Char staticBuf0[MSGSIZE0 * NUMMSGs];
static Char staticBuf1[MSGSIZE1 * NUMMSGs];

static MQASTATIC_Params poolParams0 = {staticBuf0,
    sizeof(staticBuf0), MSGSIZE0};
static MQASTATIC_Params poolParams1 = {staticBuf1,
    sizeof(staticBuf1), MSGSIZE1};

static STATICPOOL_Obj poolObj0, poolObj1;

static POOL_Obj allocators[NumAllocators] =
    {{STATICPOOL_init, (POOL_Fxns *)&STATICPOOL_FXNS,
    &poolParams0, &poolObj0}
    {{STATICPOOL_init, (POOL_Fxns *)&STATICPOOL_FXNS,
    &poolParams1, &poolObj1}};

POOL_Config POOL_config =
    {allocators, NumAllocators};
```

## POOL Manager Properties

To configure the POOL manager, the POOL\_Config structure must be defined in the application code. See “Static Configuration” on page 2-327.

The following global property must also be set in order to use the POOL module:

- ❑ **Enable POOL Manager.** If ENABLEPOOL is TRUE, each allocator specified in the POOL\_config structure (see “Static Configuration” on page 2-327) is initialized and opened.

Tconf Name: ENABLEPOOL

Type: Bool

Example:    bios.POOL.ENABLEPOOL = true;



## 2.22 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 execution.
- ❑ PRD\_tick. Advance tick counter, dispatch periodic functions.

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PRD Manager Properties and PRD Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

| Name         | Type      | Default           |
|--------------|-----------|-------------------|
| OBJMEMSEG    | Reference | prog.get("IDRAM") |
| USECLK       | Bool      | true              |
| MICROSECONDS | Int16     | 1000.0            |

#### Instance Configuration Parameters

| Name    | Type       | Default (Enum Options)    |
|---------|------------|---------------------------|
| comment | String     | "<add comments here>"     |
| period  | Int16      | 32767                     |
| mode    | EnumString | "continuous" ("one-shot") |
| fxn     | Extern     | prog.extern("FXN_F_nop")  |
| arg0    | Arg        | 0                         |
| arg1    | Arg        | 0                         |
| order   | Int16      | 0                         |

### 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 CLK Object Properties. Set the "Use On-chip Clock (CLK)" property of the PRD Manager Properties to true. Set the frequency of execution (in number of clock interrupt ticks) in the period property for the individual period object.
- ❑ **To schedule functions based on I/O availability or some other event.** Set the "Use On-chip Clock (CLK)" property of the PRD Manager Properties to false. Set the frequency of execution (in number of ticks) in the period property 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. 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 a 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 SWI priority to allow this SWI to be performed once per tick. This SWI is automatically created (or deleted) by the configuration if one or more (or no) PRD objects exist. The total time required to perform all PRD functions must be less than the number of microseconds between ticks. Any more lengthy processing should be scheduled as a separate SWI, TSK, or IDL thread.

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, and 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 in the PRD Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment containing the PRD objects.  
Tconf Name: OBJMEMSEG Type: Reference  
Example: `bios.PRD.OBJMEMSEG = prog.get("myMEM");`
- ❑ **Use CLK Manager to drive PRD.** If this property is set to true, the on-device timer hardware (managed by the CLK Module) is used to advance the tick count; otherwise, the application must invoke PRD\_tick on a periodic basis. If the CLK module is used to drive PRDs, the ticks are equal to the low-resolution time increment rate.  
Tconf Name: USECLK Type: Bool  
Example: `bios.PRD.USECLK = true;`
- ❑ **Microseconds/Tick.** The number of microseconds between ticks. If the "Use CLK Manager to drive PRD field" property above is set to true, this property is automatically set by the CLK module; otherwise, you must explicitly set this property. The total time required to perform all PRD functions must be less than the number of microseconds between ticks.  
Tconf Name: MICROSECONDS Type: Int16  
Example: `bios.PRD.MICROSECONDS = 1000.0;`

## PRD Object Properties

To create a PRD object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myPrd = bios.PRD.create("myPrd");
```

If you cannot create a new PRD object (an error occurs or the Insert PRD item is inactive in the DSP/BIOS Configuration Tool), increase the Stack Size property in the MEM Manager Properties before adding a PRD object.

The following properties can be set for a PRD object in the PRD Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this PRD object.  
Tconf Name: comment Type: String  
Example: `myPrd.comment = "my PRD";`
- ❑ **period (ticks).** The function executes after this number of ticks have elapsed.  
Tconf Name: period Type: Int16  
Example: `myPrd.period = 32767;`

- ❑ **mode.** If "continuous" is used, the function executes every "period" number of ticks. If "one-shot" is used, the function executes just once after "period" ticks.

Tconf Name: mode

Type: EnumString

Options: "continuous", "one-shot"

Example: `myPrd.mode = "continuous";`

- ❑ **function.** The function to be executed. The total time required to perform all PRD functions must be less than the number of microseconds between ticks.

Tconf Name: fxn

Type: Extern

Example: `myPrd.fxn = prog.extern("prdFxn");`

- ❑ **arg0, arg1.** Two Arg type arguments for the user-specified function above.

Tconf Name: arg0

Type: Arg

Tconf Name: arg1

Type: Arg

Example: `myPrd.arg0 = 0;`

- ❑ **period (ms).** The number of milliseconds represented by the period specified above. This is an informational property only.

Tconf Name: N/A

- ❑ **order.** Set this property to all PRD objects so that the numbers match the sequence in which PRD functions should be executed.

Tconf Name: order

Type: Int16

Example: `myPrd.order = 2;`

**PRD\_getticks***Get the current tick count***C Interface**

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <b>Syntax</b>       | <code>num = PRD_getticks();</code>               |
| <b>Parameters</b>   | Void                                             |
| <b>Return Value</b> | LgUns      num <i>/* current tick counter */</i> |
| <b>Reentrant</b>    | yes                                              |

**Description**

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 2–51, 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                                        |
| <b>Reentrant</b>    | no                                          |

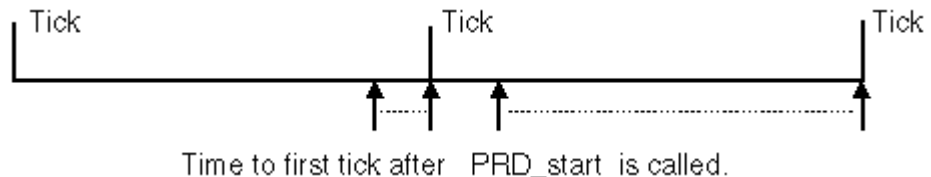
**Description**

PRD\_start starts a period object that has its mode property set to one-shot in the configuration. 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 2-7 shows, PRD ticks occur at a fixed rate and the call to PRD\_start can occur at any point between ticks

*Figure 2-7. PRD Tick Cycles*



If PRD\_start is called again before the period for the object has elapsed, the object's tick count is reset. The PRD object does not run until its "period" number of ticks have elapsed.

**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                                        |

**Reentrant**                      no

**Description**                      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.

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.

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**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, an HWI 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, 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



## 2.23 PWRM Module



The PWRM module lets you reduce the power consumption of your DSP/BIOS application.

The PWRM module is currently available for the 'C6748 EVM. See the DSP/BIOS release notes to determine which features are supported on different devices.

### Functions

- ❑ `PWRM_changeSetpoint`. Initiate a change to the V/F setpoint.
- ❑ `PWRM_configure`. Set new configuration parameters for PWRM.
- ❑ `PWRM_getCapabilities`. Get information on PWRM's capabilities on the current platform.
- ❑ `PWRM_getConstraintInfo`. Get information on constraints registered with PWRM.
- ❑ `PWRM_getCPULoad`. Get CPU load information.
- ❑ `PWRM_getCurrentSetpoint`. Get the current V/F setpoint in effect.
- ❑ `PWRM_getDependencyCount`. Get count of dependencies currently declared on a resource.
- ❑ `PWRM_getLoadMonitorInfo`. Get PWRM load monitor configuration.
- ❑ `PWRM_getNumSetpoints`. Get the number of V/F setpoints supported for the current platform.
- ❑ `PWRM_getSetpointInfo`. Get the corresponding frequency and CPU core voltage for a setpoint.
- ❑ `PWRM_getTransitionLatency`. Get the latency to scale from one setpoint to another setpoint.
- ❑ `PWRM_registerConstraint`. Register an operational constraint with PWRM.
- ❑ `PWRM_registerNotify`. Register a `pwrMNotifyFxn` function to be called on a specific power event.
- ❑ `pwrMNotifyFxn`. Function to be called for power event notification.
- ❑ `PWRM_releaseDependency`. Release a dependency that has been previously declared.
- ❑ `PWRM_resetCPULoadHistory`. Clear the CPU load history buffered by PWRM.
- ❑ `PWRM_setDependency`. Declare a dependency upon a resource.
- ❑ `PWRM_signalEvent`. Signal a power event to registered notification clients.

- ❑ PWRM\_sleepDSP. Transition the DSP to a new sleep state.
- ❑ PWRM\_startCPULoadMonitoring. Restart CPU load monitoring.
- ❑ PWRM\_stopCPULoadMonitoring. Stop CPU load monitoring.
- ❑ PWRM\_unregisterConstraint. Unregister a constraint from PWRM.
- ❑ PWRM\_unregisterNotify. Unregister for an event notification from PWRM.

**Description**

The DSP/BIOS Power Manager, PWRM, is a DSP/BIOS module that lets you reduce the power consumption of your application in the following ways:

- ❑ You can dynamically change the operating voltage and frequency of the CPU. This is called V/F scaling. Since power usage is linearly proportional to the frequency and quadratically proportional to the voltage, using the PWRM module can result in significant power savings.
- ❑ You can set custom sleep modes to save power during inactivity. These can be set statically or at run-time.
- ❑ You can coordinate sleep modes and V/F scaling using registration and notification mechanisms provided by the PWRM module.
- ❑ PWRM functions are designed to save and restore the users environment where appropriate.

For further description of these features in DSP/BIOS, see the *TMS320 DSP/BIOS User's Guide* (SPRU423).

**Constants, Types, and Structures**

```
typedef Void * PWRM_NotifyHandle;

typedef Uns PWRM_Status;

typedef struct PWRM_Config {
    Bool scaleVoltage;
    Bool waitForVoltageScale;
} PWRM_Config;

typedef struct PWRM_Attrs {
    Bool scaleVoltage;          /* scale voltage */
    Bool waitForVoltageScale; /* wait on volt change */
} PWRM_Attrs;
```

The following constants are used as return codes by various PWRM functions:

| Name                 | Usage                                                                                                                                                |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                                                                                             |
| PWRM_EFAIL           | A general failure occurred.                                                                                                                          |
| PWRM_EINVALIDEVENT   | The specified PWRM event type is invalid.                                                                                                            |
| PWRM_EINVALIDHANDLE  | The specified handle is invalid.                                                                                                                     |
| PWRM_EINVALIDPOINTER | A pointer is invalid.                                                                                                                                |
| PWRM_EINVALIDVALUE   | A value is invalid.                                                                                                                                  |
| PWRM_ENOTIMPLEMENTED | The operation is not implemented by PWRM on this platform.                                                                                           |
| PWRM_ENOTSUPPORTED   | The requested setting is not supported. For example, a client has registered with PWRM indicating that it cannot support the requested V/F setpoint. |
| PWRM_EOUTOFRANGE     | The operation could not be completed because a parameter was out of the range supported by PWRM.                                                     |
| PWRM_ETIMEOUT        | A timeout occurred while trying to complete the operation.                                                                                           |
| PWRM_ETOOMANYCALLS   | Indicates PWRM_releaseDependency has been called more times for a resource than PWRM_setDependency was called.                                       |
| PWRM_EBUSY           | The requested operation cannot be performed at this time; PWRM is busy processing a previous request.                                                |
| PWRM_EINITFAILURE    | A failure occurred while initializing V/F scaling support; V/F scaling is unavailable.                                                               |

## Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PWRM Manager Properties topic. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

### Module Configuration Parameters

| Name         | Type      | Default (Enum Options) |
|--------------|-----------|------------------------|
| ENABLE       | Bool      | false                  |
| ONCHIPMEMSEG | Reference | prog.get("IRAM")       |
| IDLECPU      | Bool      | true                   |

| Name             | Type    | Default (Enum Options)             |
|------------------|---------|------------------------------------|
| IDLEFXN          | Extern  | prog.extern("_PWRM_F_idleStopClk") |
| LOADENABLE       | Bool    | false                              |
| NUMSLOTS         | Numeric | 1                                  |
| CLKTICKSPERSLOT  | Numeric | 10                                 |
| SLOTHOOKFXN      | Extern  | prog.extern("FXN_F_nop")           |
| RESOURCETRACKING | Bool    | false                              |
| SCALING          | Bool    | false                              |
| INITCPUSP        | Numeric | 2                                  |
| INITPERSP        | Numeric | 0                                  |
| MOVESPS          | Bool    | true                               |
| SCALEVOLT        | Bool    | true                               |
| WAITVOLT         | Bool    | true                               |
| SCALECONFIGLIB   | String  | pscl_cfg_evm6748.a674              |
| VLTCROLLIB       | String  | pml_ctl_evm6748.a674               |

## PWRM Manager Properties



The following global properties can be set for the PWRM module in the PWRM Manager Properties dialog of Gconf or in a Tconf script:

- ☐ **Enable PWRM Manager.** Check this box if you want to enable the power manager. If you do not plan to use the power manager, you should leave it disabled to reduce the size of your application. You must check this box if you intend to configure other PWRM properties.

Tconf Name: ENABLE

Type: Bool

Example: `bios.PWRM.ENABLE = true;`

- ☐ **MEM section for PWRM's on-chip code/date.** Select the memory segment where PWRM should locate its code and data that must reside on-chip. The selected segment must reside in on-chip (i.e., L2 or L3) RAM. Note that the drop-down selection box in Gconf may allow you to select other, off-chip memory segments. You must select an on-chip segment, otherwise unpredictable behavior will result.

Tconf Name: ONCHIPMEMSEG

Type: Reference

Example: `bios.PWRM.DEVICEDBMEMSEG =  
          prog.get("myMEM");`

- ☐ **Idle CPU during BIOS idle loop.** Check this box to have PWRM idle the CPU during IDL loop processing. This box must be checked if you plan to use PWRM's CPU load monitoring feature.

Tconf Name: IDLECPU

Type: Bool

Example: `bios.PWRM.IDLECPU = true;`

- ☐ **Idle function.** This property configures an idle function to be called each time the idle thread passes through the idle loop. The IDLECPU property must be true to allow the IDLEFXN property to be defined. You can assign your own idle function or the `_PWRM_F_idleStopClk` function provided by the PWRM module. The `_PWRM_F_idleStopClk` function sets the PDCCMD register to stop GEM clocks (without a full transition of GEM to standby), performs steps necessary to monitor the CPU load, and executes the idle instruction.

The idle function signature is:

`Void idleFxn(Void)`

Tconf Name: IDLEFXN

Type: Extern

Example: `bios.PWRM.IDLEFXN =  
prog.extern("_PWRM_F_idleStopClk");`

- ☐ **Enable CPU Load Monitoring.** Check this box if you want PWRM to measure and accumulate CPU load. The "Idle CPU during BIOS idle loop" box must be checked for this box to be writeable.

Tconf Name: LOADENABLE

Type: Bool

Example: `bios.PWRM.LOADENABLE = true;`

- ☐ **Number of history slots to buffer.** This property allows you to specify the number of "slots" or intervals of CPU load history to be buffered within PWRM. The "Enable CPU Load Monitoring" box must be checked for this property to be writeable.

Tconf Name: NUMSLOTS

Type: Numeric

Example: `bios.PWRM.NUMSLOTS = 5;`

- ☐ **Number of CLK ticks per slot.** This property allows you to specify the duration of CPU load history slots, in terms of CLK clock ticks. For example, if the CLK module is configured for 1 tick per millisecond, then a value of 10 for this property means CPU history slot has a duration of 10 milliseconds. The "Enable CPU Load Monitoring" box must be checked for this box to be writeable.

Tconf Name: CLKTICKSPERSLOT

Type: Numeric

Example: `bios.PWRM.CLKTICKSPERSLOT = 10;`

- ❑ **Hook function to call upon slot finalization.** This property allows you to configure a function to be called upon the finalization of each history slot. This function is called from a SWI context (so should not block), and will be called as the last step of finalization, after the slot data has been written to PWRM's internal history buffer. The "Enable CPU Load Monitoring" box must be checked for this property to be writeable.

The hook function signature is:

```
Void slotHookFxn(Uns arg0)
```

When this function is called, arg0 indicates the slot finalization timestamp.

Tconf Name: SLOTHOOKFXN Type: Extern

Example: `bios.PWRM.SLOTHOOKFXN =  
          prog.extern("myHookFunction");`

- ❑ **Enable Resource Tracking.** Check this box if you want to enable PWRM support for dynamic resource tracking.

Tconf Name: RESOURCETRACKING Type: Bool

Example: `bios.PWRM.RESOURCETRACKING = true;`

- ❑ **Enable V/F Scaling Support.** Check this box to enable PWRM's support for V/F scaling.

Tconf Name: SCALING Type: Bool

Example: `bios.PWRM.SCALING = true;`

- ❑ **Initial CPU domain setpoint.** This property allows you to specify the initial setpoint for the CPU domain after booting. For details, see the PWRM\_changeSetpoint description. The "Enable V/F Scaling Support" box must be checked for this field to be writeable.

Tconf Name: INITCPUSP Type: Numeric

Example: `bios.PWRM.INITCPUSP = 2;`

- ❑ **Initial PER domain setpoint.** This property allows you to specify the initial setpoint for the peripheral domain after booting. For details, see the PWRM\_changeSetpoint description. The "Enable V/F Scaling Support" box must be checked for this field to be writeable.

Tconf Name: INITPERSP Type: Numeric

Example: `bios.PWRM.INITPERSP = 0;`

- ❑ **Move to these setpoints during PWRM initialization.** This property allows you to specify whether or not PWRM should move (i.e., change voltage and frequency) to the CPU and PER domain setpoints during the boot and initialization process. If this box is not checked, the setpoint values are used only to establish initial hardware state information for PWRM by initializing internal data

structures. If this box is checked, PWRM additionally scales the hardware to these initial setpoints when PWRM initializes. Note that this scaling is done as part of the boot process; clients are not notified before/after the scaling, as is possible with the PWRM\_changeSetpoint API. The "Enable V/F Scaling Support" box must be checked for this box to be writeable.

Tconf Name: MOVESPS

Type: Bool

Example: `bios.PWRM.MOVESPS = true;`

- ❑ **Scale voltage along with frequency.** This property specifies whether voltage should be scaled along with frequency. You may want to disable voltage scaling to reduce latency when changing the frequency. If this property is set to true, a change to the frequency (via PWRM\_changeSetpoint) results in a voltage change when possible. If this property is set to false, voltage is not scaled down along with frequency. The voltage is always scaled up if the new setpoint frequency is higher than that supported at the current voltage. This property also controls whether voltage is reduced during sleep modes that support voltage reduction. This setting can be modified at runtime using the PWRM\_configure function. The "Enable V/F Scaling Support" box must be checked for this box to be writeable.

Tconf Name: SCALEVOLT

Type: Bool

Example: `bios.PWRM.SCALEVOLT = true;`

- ❑ **Wait while voltage is being scaled down.** This property specifies whether PWRM functions should wait during down-voltage transitions. Such transition times can be long, as they typically depend upon power supply load. Currently, it is recommended that this property remain set to true. (Note that the PWRM module always waits during up-voltage transitions; this is required to avoid over-clocking the DSP.) This setting can be modified at runtime using the PWRM\_configure function. The "Enable V/F Scaling Support" box must be checked for this box to be writeable.

Tconf Name: WAITVOLT

Type: Bool

Example: `bios.PWRM.WAITVOLT = true;`

- ❑ **Scaling configuration library.** This property allows you to specify the scaling configuration library to link with. Specify only the filename of the library to link with for this property. The include path to the configuration library should be added to the linker command file if it is not in the default path. The "Enable V/F Scaling Support" box must be checked for this box to be writeable.

Tconf Name: SCALECONFIGLIB

Type: String

Example:    `bios.PWRM.SCALECONFIGLIB =  
              "pscl_cfg_evm6748.a674";`

- ❑ **Voltage control library.** This property allows you to specify the board-specific voltage control library to link with. Specify only the filename of the library to link with for this property. The include path to the control library should be added to the linker command file if it is not in the default path. The "Enable V/F Scaling Support" box must be checked for this box to be writeable.

Tconf Name: VLTCONTROLLIB

Type: String

Example:    `bios.PWRM.VLTCONTROLLIB =  
              "pmi_ctl_evm6748.a674";`



**PWRM\_changeSetpoint**

*Initiate a change to the V/F setpoint*

**C Interface**

|                     |                                                                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | status = PWRM_changeSetpoint(domain, newSetpoint, notifyTimeout);                                                                                      |
| <b>Parameters</b>   | PWRM_Domain domain; /* setpoint domain */<br>Uns newSetpoint; /* new V/F setpoint */<br>Uns notifyTimeout; /* maximum time to wait for notification */ |
| <b>Return Value</b> | PWRM_Status status; /* returned status */                                                                                                              |
| <b>Reentrant</b>    | yes                                                                                                                                                    |

**Description**



PWRM\_changeSetpoint changes the voltage and frequency of a scalable domain.

Reducing the clock rate (frequency) results in a linear decrease in power consumption. Reducing the operating voltage results in a quadratic reduction in power consumption. Note that there are issues you should be aware of when reducing the clock frequency. For a discussion of these issues, see the *TMS320 DSP/BIOS User's Guide* (SPRA423).

The domain parameter indicates the relevant scaling domain. The domains that can be scaled vary by platform, and are enumerated as PWRM\_Domain. For example, for the TMS320C6748, there are two scalable domains: the CPU domain (PWRM\_CPU) and the peripheral domain (PWRM\_PER).

The newSetpoint parameter is a numeric value that indexes into a table of frequency/voltage pairs, as defined by the underlying scaling configuration library. For example, the following table shows the out-of-the-box setpoints for the 'C6748 EVM:

| Setpoint | Frequency (MHz) | Voltage (volts) |
|----------|-----------------|-----------------|
| 2        | 300             | 1.2             |
| 1        | 200             | 1.1             |
| 0        | 100             | 1.0             |

The notifyTimeout parameter is the maximum amount of time (in system clock ticks) to wait for registered notification functions (set by PWRM\_registerNotify) to respond to a delayed completion, before declaring failure and returning PWRM\_ETIMEOUT.

For example, if notifyTimeout is set to 200, PWRM\_changeSetpoint waits up to 200 ticks (typically 200 milliseconds) before declaring that a function has failed to respond. PWRM uses notifyTimeout for each type of notification. For example, if notification functions are registered for both before and after setpoint changes, PWRM\_changeSetpoint waits up to notifyTimeout on each type of notification. All registered notification functions are called from the context of PWRM\_changeSetpoint.

PWRM\_changeSetpoint returns one of the following constants as a status value of type PWRM\_Status:

| Name                | Usage                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| PWRM_SOK            | The operation succeeded and the new setpoint is in effect.                                                                                        |
| PWRM_EFAIL          | A general failure occurred. The requested setpoint transition did not occur.                                                                      |
| PWRM_NOTIMPLEMENTED | V/F scaling is not implemented by PWRM on this platform.                                                                                          |
| PWRM_ENOTSUPPORTED  | The operation could not be completed because a client registered a constraint with PWRM indicating that it cannot support the requested setpoint. |
| PWRM_EOUTOFRANGE    | The operation could not be completed because domain or newSetpoint is not a valid value for the platform.                                         |
| PWRM_ETIMEOUT       | A registered notification function did not complete its processing within the specified notifyTimeout.                                            |
| PWRM_EBUSY          | The requested operation cannot be performed at this time; PWRM is busy processing a previous request.                                             |
| PWRM_EINITFAILURE   | A failure occurred while initialization V/F scaling support; V/F scaling is unavailable.                                                          |

The application should treat return values of PWRM\_ETIMEOUT or PWRM\_EFAIL as critical system failures. These values indicate the notification client is taking too long or is unresponsive, and the system is in an unknown state.

PWRM\_changeSetpoint disables SWI and TSK scheduling when it begins making a change. However, HWIs may run during the notification process. After the setpoint has been changed, SWI and TSK scheduling is re-enabled, and a context switch occurs only if some other thread has since been made ready to run.

## Constraints and Calling Context

- ❑ PWRM\_changeSetpoint cannot be called from an HWI.
- ❑ This API cannot be called from a program's main() function.
- ❑ PWRM\_changeSetpoint can be called from a SWI only if notifyTimeout is 0.

## Example

```
#define TIMEOUT    10  /* timeout for notifications */

PWRM_Status status;
Uns i = 5;

status = PWRM_changeSetpoint(PWRM_CPU, i, TIMEOUT);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "New setpoint = %d", i);
}
else if (status == PWRM_ENOTSUPPORTED) {
    LOG_printf(TRACE, "Setpoint %d unsupported", i);
}
else {
    LOG_printf(TRACE, "Error: status = %x", status);
}
GBL_getFrequency
GBL_setFrequency
```

**PWRM\_configure***Set new configuration properties for PWRM***C Interface**

|                     |                                                                             |
|---------------------|-----------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = PWRM_configure(attrs);</code>                                |
| <b>Parameters</b>   | <code>PWRM_Attrs    attrs;            /* configuration attributes */</code> |
| <b>Return Value</b> | <code>PWRM_Status   status;           /* returned status */</code>          |

**Reentrant**                                yes

**Description**

PWRM\_configure specifies new configuration properties for the PWRM module. It overrides those specified in the static configuration.

Configuration parameters are specified via a PWRM\_Attrs structure. This attribute structure can vary by platform. For the TMS320C6748, this structure contains the following:

```
typedef struct PWRM_Attrs {
    Bool scaleVoltage;           /* scale voltage */
    Bool waitForVoltageScale; /* wait for down ramp */
} PWRM_Attrs;
```

In this structure, scaleVoltage indicates whether PWRM should scale voltages during setpoint changes and switching to sleep modes that support voltage reduction. It corresponds to the "Scale voltage along with frequency" configuration property. If scaleVoltage is TRUE, the voltage is scaled down if possible when going to a lower frequency or activating sleep modes. If scaleVoltage is FALSE, the voltage is not scaled lower. The voltage is always scaled up if the new (destination) setpoint frequency is higher than that supported at the current voltage.

The waitForVoltageScale flag indicates whether PWRM should wait for a down-voltage transition to complete before returning from PWRM\_changeSetpoint. It corresponds to the "Wait while voltage is being scaled down" configuration property. Such transition times can be long, as they typically depend upon power supply load. Currently, it is recommended that this item always be TRUE. (The PWRM module always waits during up-voltage transitions; this is required to avoid over-clocking the DSP.)

PWRM\_configure returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                          |
|--------------------|----------------------------------------------------------------|
| PWRM_SOK           | The operation succeeded.                                       |
| PWRM_EINVALIDVALUE | The operation failed because one of the attributes is invalid. |

### Example

```
PWRM_Attrs pmAttrs;  
  
pmAttrs.scaleVoltage = TRUE;  
pmAttrs.waitForVoltageScale = TRUE;  
  
status = PWRM_configure(pmAttrs);  
if (status != PWRM_SOK) {  
    LOG_printf(TRACE, "ERROR: status = %x", status);  
}
```

PWRM\_getCapabilities

Get information on PWRM capabilities on the current platform

C Interface

|              |                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------|
| Syntax       | status = PWRM_getCapabilities(capsMask);                                                               |
| Parameters   | Uns                    *capsMask;   /* pointer to location for capabilities */                         |
| Return Value | PWRM_Status status;           /* returned status */                                                    |
| Reentrant    | yes                                                                                                    |
| Description  | PWRM_getCapabilities returns information about the PWRM module's capabilities on the current platform. |



The capsMask parameter should point to the location where PWRM\_getCapabilities should write a bitmask that defines the capabilities. You can use the following constants to check for capabilities in the bitmask:

| Name                   | Usage                                                   |
|------------------------|---------------------------------------------------------|
| PWRM_CLOADMONITORING   | The PWRM module supports CPU load monitoring.           |
| PWRM_CRESOURCETRACKING | The PWRM module supports dynamic resource tracking.     |
| PWRM_CVFSCALING        | The PWRM module supports voltage and frequency scaling. |

PWRM\_getCapabilities returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                         |
|----------------------|---------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                      |
| PWRM_EINVALIDPOINTER | The operation failed because the capsMask parameter was NULL. |

**Example**

```
PWRM_Status status;
Uns capsMask;

/* Query PWRM capabilities on this platform */
status = PWRM_getCapabilities(&capsMask);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "Caps mask=0x%X", capsMask);
    if ((capsMask & PWRM_CVFSCALING) == 0) {
        LOG_printf(TRACE, "V/F scaling not supported!");
    }
}
else {
    LOG_printf(TRACE, "ERROR: status = %x", status);
}
```

PWRM\_getConstraintInfo

Get information on constraints registered with PWRM

C Interface

|              |                                                                                                                                                                 |         |                                     |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|-------------------------------------|
| Syntax       | status = PWRM_getConstraintInfo(type, value);                                                                                                                   |         |                                     |
| Parameters   | PWRM_Constraint                                                                                                                                                 | type    | /* type of constraint */            |
|              | Arg                                                                                                                                                             | *value  | /* type-specific constraint mask */ |
| Return Value | PWRM_Status                                                                                                                                                     | status; | /* returned status */               |
| Reentrant    | yes                                                                                                                                                             |         |                                     |
| Description  | PWRM_getConstraintInfo returns information about constraints that are currently registered with the PWRM module, via previous calls to PWRM_registerConstraint. |         |                                     |



The type parameter identifies the type of constraint being queried. The constraint types vary by platform, and are enumerated as PWRM\_Constraint. For example, for the TMS320C6748, the available constraints are the following:

| Name                              | Usage                                              |
|-----------------------------------|----------------------------------------------------|
| PWRM_DISALLOWEDSLEEPSTATE_MASK    | Sleep states that should not be activated.         |
| PWRM_DISALLOWED_CPU_SETPOINT_MASK | CPU setpoints that should not be activated.        |
| PWRM_DISALLOWED_PER_SETPOINT_MASK | Peripheral setpoints that should not be activated. |

See PWRM\_registerConstraint for a description of these masks.

PWRM\_getContraintInfo returns a value that is the aggregate of all constraints that are currently registered. For example, if one PWRM client disallowed the sleep state PWRM\_SLEEP, and a different client disallowed the sleep state PWRM\_DEEPSLEEP, the returned value will be (PWRM\_SLEEP | PWRM\_DEEPSLEEP).



This function returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                 |
|----------------------|---------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                              |
| PWRM_EINVALIDVALUE   | The operation failed because type does not correspond to a supported constraint type. |
| PWRM_EINVALIDPOINTER | The operation failed because the value parameter was NULL.                            |

### Example

```

/* aggregate constraints */
Arg value;

status = PWRM_getConstraintInfo(
    PWRM_DISALLOWED_CPU_SETPOINT_MASK, &value);

if (status == PWRM_SOK) {
    LOG_printf(TRACE, "Disallowed CPU setpoint
mask=0x%x", value);
}
else {
    LOG_printf(TRACE, "Error: status=%x", status);
}

```

**PWRM\_getCPULoad***Get CPU load information***C Interface**

|                     |                                                                                                                                                         |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = PWRM_getCPULoad(numSlots, loadInfo);</code>                                                                                              |
| <b>Parameters</b>   | <code>Uns numSlots; /* # of history slots of load info to get */</code><br><code>PWRM_CPULoadInfo *loadInfo; /* array of load info structures */</code> |
| <b>Return Value</b> | <code>PWRM_Status status; /* returned status */</code>                                                                                                  |
| <b>Reentrant</b>    | yes                                                                                                                                                     |

**Description**

PWRM\_getCPULoad reports CPU load information accumulated by the PWRM module. Load history is accumulated in "slots" of a configured duration. (See "PWRM Manager Properties" on page 2-342" for details on configuring the number of slots to be buffered by PWRM and the duration of those slots.)

The PWRM\_CPULoadInfo structure reports the total number of CPU cycles for the slot, the number of those cycles where the CPU was busy, and a timestamp indicating when the slot was finalized (completed).

```
typedef struct PWRM_CPULoadInfo {
    Uns busyCycles; /* # of cycles CPU was busy */
    Uns totalCycles; /* total # of CPU cycles in slot */
    Uns timeStamp; /* time when slot finalized */
} PWRM_CPULoadInfo;
```

The numSlots parameter specifies how many history slots of the loadInfo array should be written by PWRM. History slots are reported in last-in, first-out (LIFO) order. In other words, the most recently finalized history slot is reported in the first element of the loadInfo array, the slot previous to that is reported in the second element, and so on.

PWRM maintains history slots as a ring buffer. Once all history slots have been filled, finalizing the next slot causes the oldest history slot to be overwritten. The number of slots buffered in PWRM is configured statically. If numSlots is greater than the number of slots maintained by PWRM, then a PWRM\_EOUTOFRANGE error is reported, and no history data is copied to loadInfo.

If a history slot is "empty" then the slot data elements have a default (reset) value:

```
PWRM_CPULoadInfo _PWRM_resetLoad = {
    0,    /* busyCycles */
    0,    /* totalCycles */
    0     /* timestamp */
};
```

Slots are empty if not enough time has passed for PWRM's internal slot buffer to fill, either since startup, or since a call to PWRM\_resetCPULoadHistory.

PWRM only reports load information for finalized slots. In other words, PWRM does not report information on the currently filling but not yet completed slot. It is only when the current slot is finalized that the accumulated busy and total cycles will be stored in PWRM's ring buffer.

PWRM\_getCPULoad returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                                                   |
|----------------------|-------------------------------------------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                                                                |
| PWRM_EINITFAILURE    | The operation failed because there was a failure during load monitor initialization.                                    |
| PWRM_ENOTSUPPORTED   | The operation failed because load monitoring is not enabled.                                                            |
| PWRM_EOUTOFRANGE     | The operation failed because numSlots is greater than the number of history slots PWRM has been configured to maintain. |
| PWRM_EINVALIDPOINTER | The operation failed because the loadInfo parameter was NULL.                                                           |

### Example

```
#define NUMSLOTS 5

PWRM_CPULoadInfo history[NUMSLOTS];

status = PWRM_getCPULoad(NUMSLOTS, history);
if (status == PWRM_SOK) {
    displayCPULoad(NUMSLOTS, history);
}
else {
    LOG_printf	TRACE, "Error: status = %x", status;
}
```

PWRM\_getCurrentSetpoint

Get the current setpoint for a domain

C Interface

|              |                                                                                                             |
|--------------|-------------------------------------------------------------------------------------------------------------|
| Syntax       | status = PWRM_getCurrentSetpoint(domain, setpoint);                                                         |
| Parameters   | PWRM_Domain domain /* setpoint domain */<br>Uns *setpoint; /* current V/F setpoint */                       |
| Return Value | PWRM_Status status; /* returned status */                                                                   |
| Reentrant    | no                                                                                                          |
| Description  | PWRM_getCurrentSetpoint returns the V/F scaling setpoint currently in use for the specified scaling domain. |



The domain parameter indicates the relevant scaling domain. The domains that can be scaled vary by platform, and are enumerated as PWRM\_Domain.

The setpoint parameter should point to the location where PWRM\_getCurrentSetpoint should write the current setpoint.Refer to the DSP/BIOS release notes for the default setpoints supported by the default scaling configuration library.

PWRM\_getCurrentSetpoint returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                  |
|----------------------|----------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                               |
| PWRM_EINVALIDPOINTER | The operation failed because the setpoint parameter was NULL.                          |
| PWRM_EINVALIDVALUE   | The operation failed because the domain parameter is invalid.                          |
| PWRM_EINITFAILURE    | A failure occurred while initializing V/F scaling support; V/F scaling is unavailable. |
| PWRM_ENOTSUPPORTED   | The operation failed because V/F scaling is not enabled.                               |

- Constraints and Calling Context
- ❑

If a call to PWRM\_getCurrentSetpoint is made during a change to the current setpoint, the value PWRM\_getCurrentSetpoint returns may be the old setpoint and not the new setpoint.

**Example**

```
PWRM_Status status;
Uns currSetpoint;

status = PWRM_getCurrentSetpoint(PWRM_CPU,
                                &currSetpoint);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "Setpoint: %d", currSetpoint);
}
else {
    LOG_printf(TRACE, "ERROR: status = %x", status);
}
```

PWRM\_getDependencyCount

Get count of dependencies declared on a resource

C Interface

|              |                                                      |             |                                         |
|--------------|------------------------------------------------------|-------------|-----------------------------------------|
| Syntax       | status = PWRM_getDependencyCount(resourceID, count); |             |                                         |
| Parameters   | Uns                                                  | resourceID; | /* resource ID */                       |
|              | Uns                                                  | *count;     | /* pointer to where count is written */ |
| Return Value | PWRM_Status                                          | status;     | /* returned status */                   |
| Reentrant    | yes                                                  |             |                                         |

Description



PWRM\_getDependencyCount returns the number of dependencies that are currently declared on a resource. Normally this corresponds to the number of times PWRM\_setDependency has been called for the resource, minus the number of times PWRM\_releaseDependency has been called for the same resource. On some platforms, early boot code may enable individual resources before PWRM initializes. To ensure proper state synchronization in this case, PWRM queries individual resource states as the PWRM module initializes. So, it is possible for the returned "count" value to be 1, even if PWRM\_setDependency has not been called by the application.

Resource IDs are device-specific, and are defined in a PWRM\_Resource enumeration in a device-specific header file.

Resource IDs are device-specific. They are defined in a PWRM\_Resource enumeration in a device-specific header file. For an example, see pwr6748.h for the TMS320C6748 DSP.

PWRM\_getDependencyCount returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                             |
|----------------------|---------------------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded, and the reference count was written to the location pointed to by count. |
| PWRM_EINVALIDPOINTER | The operation failed because the count parameter was NULL.                                        |
| PWRM_EINVALIDVALUE   | The operation failed because PWRM cannot control this resource.                                   |
| PWRM_ENOTSUPPORTED   | The operation failed because resource tracking is not enabled.                                    |
| PWRM_EOUTOFRANGE     | The specified resourceID is outside the range of valid resource IDs.                              |

**Example**

```
#include <pwrn.h>
#include <pwrn6748.h>
...

/* Display some dependency counts */
LOG_printf(TRACE, "Current dependencies:");

status = PWRM_getDependencyCount(PWRM_RSRC_UART_0,
                                &count);

if (status == PWRM_SOK) {
    LOG_printf(TRACE, "UART_0 dependency count = %d",
              count);
}
else {
    LOG_printf(TRACE, "ERROR: status = %x", status);
}
```

PWRM\_getLoadMonitorInfo

Get PWRM load monitor configuration

C Interface

|              |                                                               |
|--------------|---------------------------------------------------------------|
| Syntax       | status = PWRM_getLoadMonitorInfo(monitorInfo);                |
| Parameters   | PWRM_LoadMonitorInfo *monitorInfo; /* pointer to structure */ |
| Return Value | PWRM_Status status; /* returned status */                     |
| Reentrant    | yes                                                           |

**Description** PWRM\_getLoadMonitorInfo returns information about PWRM's load monitor configuration, in the structure pointed to by monitorInfo.



```
typedef struct PWRM_LoadMonitorInfo {
    Uns numSlots; /* # of history slots */
    Uns ticksPerSlot; /* # of CLK ticks per slot */
} PWRM_LoadMonitorInfo;
```

PWRM\_getLoadMonitorInfo returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                            |
|----------------------|------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                         |
| PWRM_EINVALIDPOINTER | The operation failed because the monitorInfo parameter was NULL. |
| PWRM_ENOTSUPPORTED   | The operation failed because CPU load monitoring is not enabled. |

Example

```
PWRM_LoadMonitorInfo monitorInfo;
status = PWRM_getLoadMonitorInfo(&monitorInfo);
if (status == PWRM_SOK) {
    LOG_printf(TRACE, "numSlots=%d, ticksPerSlot=%d",
        monitorInfo.numSlots, monitorInfo.ticksPerSlot);
}
else {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```



**PWRM\_getNumSetpoints***Get number of setpoints for a domain***C Interface****Syntax**

```
status = PWRM_getNumSetpoints(domain, numberSetpoints);
```

**Parameters**

```
PWRM_Domain domain;           /* setpoint domain */
Uns          *numberSetpoints; /* number of supported setpoints */
```

**Return Value**

```
PWRM_Status status;           /* returned status */
```

**Reentrant**

yes

**Description**

PWRM\_getNumSetpoints returns the number of setpoints supported by the specified domain.



The domain parameter indicates the relevant scaling domain. The domains that can be scaled vary by platform, and are enumerated as PWRM\_Domain.

The numberSetpoints parameter should point to the location where PWRM\_getNumSetpoints should write the number of setpoints. Refer to the DSP/BIOS release notes for the default setpoints supported by the default scaling configuration library.

PWRM\_getNumSetpoints returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                 |
|----------------------|---------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                              |
| PWRM_EINVALIDPOINTER | The operation failed because the numberSetpoints parameter was NULL.                  |
| PWRM_EINVALIDVALUE   | The operation failed because the domain parameter is invalid.                         |
| PWRM_EINITFAILURE    | A failure occurred while initializing V/F scaling support; V/F scaling is unavailable |
| PWRM_ENOTSUPPORTED   | The operation failed because V/F scaling is not enabled.                              |

**Example**

```
PWRM_Status status;
Uns numSetpoints;

status = PWRM_getNumSetpoints(domain, &numSetpoints);
if (status == PWRM_SOK) {
    LOG_printf	TRACE, "NumSetpoints: %d", numSetpoints);
}
else {
    LOG_printf	TRACE, "Error: status = %x", status);
}
```

### Get frequency and CPU core voltage for a setpoint

## C Interface

**Syntax** `status = PWRM_getSetpointInfo(domain, setpoint, frequency, voltage);`

|                   |             |             |                               |
|-------------------|-------------|-------------|-------------------------------|
| <b>Parameters</b> | PWRM_Domain | domain;     | /* setpoint domain */         |
|                   | Uns         | setpoint;   | /* the setpoint to query */   |
|                   | Uns         | *frequency; | /* frequency (in kHz) */      |
|                   | Uns         | *voltage;   | /* voltage (in millivolts) */ |

**Return Value** PWRM\_Status status; /\* returned status \*/

Reentrant yes

|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| <b>Description</b> | PWRM_getSetpointInfo returns the frequency and voltage for a given setpoint. |
|--------------------|------------------------------------------------------------------------------|



The domain parameter indicates the relevant scaling domain for which the setpoint information is to be retrieved. The domains that can be scaled vary by platform, and are enumerated as PWRM\_Domain.

The setpoint parameter should specify the setpoint value for which you want to know the frequency and voltage on this platform. Refer to the DSP/BIOS release notes for the default setpoints supported by the default scaling configuration library.

The frequency parameter should point to the location where PWRM\_getSetpointInfo should write the frequency for the specified setpoint.

The voltage parameter should point to the location where PWRM\_getSetpointInfo should write the voltage for the specified setpoint.

PWRM\_getSetpointInfo returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                     |
|----------------------|---------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                  |
| PWRM_EINVALIDVALUE   | The operation failed because the domain parameter is invalid.             |
| PWRM_EINVALIDPOINTER | The operation failed because the frequency or voltage parameter was NULL. |

| Name               | Usage                                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------------------|
| PWRM_EINITFAILURE  | A failure occurred while initializing V/F scaling support; V/F scaling is unavailable                            |
| PWRM_ENOTSUPPORTED | The operation failed because V/F scaling is not enabled.                                                         |
| PWRM_EOUTOFRANGE   | The operation failed because the setpoint parameter is out of range of valid setpoints for the specified domain. |

**Example**

```
PWRM_Status status;

/* locations for saving CPU setpoint info */
Uns freq;
Uns volts;

status = PWRM_getSetpointInfo(PWRM_CPU, 0,
                              &freq, &volts);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status=%x", status);
}
```

**PWRM\_getTransitionLatency***Get latency to scale between specific setpoints***C Interface****Syntax**

```
status = PWRM_getTransitionLatency(domain, initialSetpoint,
                                   finalSetpoint, frequencyLatency, voltageLatency);
```

**Parameters**

```
PWRM_Domain domain;           /* setpoint domain */
Uns          initialSetpoint;  /* setpoint to be scaled from */
Uns          finalSetpoint;    /* setpoint to be scaled to */
Uns          *frequencyLatency; /* frequency transition latency */
Uns          *voltageLatency;  /* voltage transition latency */
```

**Return Value**

```
PWRM_Status status;          /* returned status */
```

**Reentrant**

```
yes
```

**Description**

PWRM\_getTransitionLatency retrieves the latencies (times required) in microseconds to scale from a specific setpoint to another specific setpoint.



The domain parameter should indicate the relevant scaling domain.

The initialSetpoint parameter should specify the setpoint from which the transition would start. The finalSetpoint parameter should specify the setpoint at which the transition would end. See PWRM\_changeSetpoint and the DSP/BIOS release notes for a list of valid domains and setpoints.

The frequencyLatency parameter should point to the location where PWRM\_getTransitionLatency should write the time required to change the frequency from that of the initialSetpoint to that of the finalSetpoint in microseconds.

Similarly, the voltageLatency should point to the location where PWRM\_getTransitionLatency should write the time required to change the voltage from that of the initialSetpoint to that of the finalSetpoint in microseconds.

When frequency and voltage are scaled together, the total latency is the sum of the frequency scaling latency and the voltage scaling latency.

PWRM\_getTransitionLatency returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                        |
|----------------------|----------------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                                     |
| PWRM_EFAIL           | A general failure occurred.                                                                  |
| PWRM_EINVALIDVALUE   | The operation failed because the domain, initialSetpoint or finalSetpoint value was invalid. |
| PWRM_EINVALIDPOINTER | The operation failed because the frequencyLatency or voltageLatency parameter was NULL.      |
| PWRM_EINITFAILURE    | A failure occurred while initializing V/F scaling support; V/F scaling is unavailable        |
| PWRM_ENOTIMPLEMENTED | The operation failed because V/F scaling is not supported.                                   |

The time required to change a setpoint may not be deterministic (depending on the hardware characteristics, the underlying software implementation, and the specific V/F swing), but it is bounded by the value returned by PWRM\_getTransitionLatency.

### Example


```
PWRM_Status status;
Uns frequencyLatency;
Uns voltageLatency;

status = PWRM_getTransitionLatency(PWRM_CPU, 15, 0,
                                   &frequencyLatency, &voltageLatency);

if (status != PWRM_SOK) {
    LOG_printf	TRACE, "Error: status=%x", status);
}
else {
    LOG_printf	TRACE, "Frequency latency: %d, Voltage
latency: %d",
               frequencyLatency, voltageLatency);
}
```

Reentrant yes

**Description**



Before taking certain actions, PWRM checks to see if the requested action would conflict with a client-registered constraint. If the action does conflict, PWRM will not proceed with the request. PWRM\_registerConstraint is the API that allows clients to register their constraints with PWRM.



The type parameter identifies the type of constraint being registered. The constraint types vary by platform, and are enumerated as `PWRM_Constraint`. For example, for the TMS320C6748, the available constraints are the following:

| Name                              | Usage                                              |
|-----------------------------------|----------------------------------------------------|
| PWRM_DISALLOWED_SLEEPSTATE_MASK   | Sleep states that should not be activated.         |
| PWRM_DISALLOWED_CPU_SETPOINT_MASK | CPU setpoints that should not be activated.        |
| PWRM_DISALLOWED_PER_SETPOINT_MASK | Peripheral setpoints that should not be activated. |

The PWRM\_DISALLOWEDSLEEPSTATE\_MASK is a bitmask of the sleep states that can be activated via PWRM\_sleepDSP. For the TMS320C6748, the states are identified via a combination of the following bitmask values: PWRM\_STANDBY, PWRM\_SLEEP, and PWRM\_DEEPSLEEP.

The `PWRM_DISALLOWED_CPU_SETPOINT_MASK` is a bitmask of setpoints that the CPU should not be transitioned to. For example, if a driver won't operate properly below a specific CPU performance level, it can register the lower performance levels via the setpoint mask. Setpoint IDs are zero-based, with a lower index representing a lower performance

level. They are represented right-justified in the bitmask. For example, to disallow setpoint "0", the mask is "0x1". To disallow the lowest five setpoints the mask value is "0x1F".

The PWRM\_DISALLOWED\_PER\_SETPOINT\_MASK performs the same function as the PWRM\_DISALLOWED\_CPU\_SETPOINT\_MASK, but corresponds to scaling of the peripheral domain. The bitmask follows the same format; setpoints are zero-based and have right-justified bit positions, starting with "0x1".

It is important that clients call PWRM\_unregisterConstraint when the operational constraint no longer exists. Otherwise, PWRM may be left unnecessarily restricted from activating power savings.

PWRM\_registerConstraint returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                 |
|----------------------|---------------------------------------------------------------------------------------|
| PWRM_SOK             | The operation succeeded.                                                              |
| PWRM_EFAIL           | The operation failed due to a memory allocation failure.                              |
| PWRM_EINVALIDVALUE   | The operation failed because type does not correspond to a supported constraint type. |
| PWRM_EINVALIDPOINTER | The operation failed because the handle parameter was NULL.                           |

Constraints and Calling Context

- ❑ PWRM\_registerConstraint cannot be called from an SWI or HWI. This is because PWRM\_registerConstraint internally calls MEM\_alloc, which may cause a context switch.

Example

```
/* constraint handle */
PWRM_ConstraintHandle handle;

status = PWRM_registerConstraint(
    PWRM_DISALLOWED_CPU_SETPOINT_MASK,
    0x7, &handle);

if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: registration status=%x",
status);
}
```



## C Interface

The `eventMask` parameter is an event-specific mask. Currently the `eventMask` is not used by PWRM.

The `notifyFxn` parameter specifies the function to call when the specified power event occurs. The `notifyFunction` must behave as described in the `pwrMNotifyFxn` section.

The `clientArg` parameter is an arbitrary argument to be passed to the client upon notification. This argument may allow one notify function to be used by multiple instances of a driver (that is, the `clientArg` can be used to identify the instance of the driver that is being notified).

The `notifyHandle` parameter should point to the location where `PWRM_registerNotify` should write a notification handle. If the application later needs to unregister the notification function, the application should pass this handle to `PWRM_unregisterNotify`.

The `delayedCompletionFxn` is a pointer to a function provided by the PWRM module to the client at registration time. If a client cannot act immediately upon notification, its notify function should return `PWRM_NOTIFYNOTDONE`. Later, when the action is complete, the client should call the `delayedCompletionFxn` to signal PWRM that it has finished. The `delayedCompletionFxn` is a void function, taking no arguments, and having no return value. If a client can and does act immediately on the notification, it should return `PWRM_NOTIFYDONE` in response to notification, and should not call the `delayedCompletionFxn`.

For example, if a DMA driver is to prepare for a setpoint change, it may need to wait for the current DMA transfer to complete. When the driver's DMA completes (for example, on the next hardware interrupt), it calls the `delayedCompletionFxn` function provided when it registered for notification. This completion function tells the PWRM module that the driver is finished. Meanwhile, the PWRM module was able to continue notifying other clients, and was waiting for all clients to signal completion.

`PWRM_registerNotify` returns one of the following constants as a status value of type `PWRM_Status`:

| Name                              | Usage                                                                                                                                                        |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>PWRM_SOK</code>             | The function was successfully registered.                                                                                                                    |
| <code>PWRM_EFAIL</code>           | The operation failed due to a memory allocation failure.                                                                                                     |
| <code>PWRM_EINVALIDPOINTER</code> | The operation failed because the <code>notifyFxn</code> , <code>notifyHandle</code> , or <code>delayedCompletionFxn</code> parameter was <code>NULL</code> . |
| <code>PWRM_EINVALIDEVENT</code>   | Operation failed because <code>eventType</code> is invalid.                                                                                                  |

## Constraints and Calling Context

- ❑ PWRM\_registerNotify cannot be called from a SWI or HWI. This is because PWRM\_registerNotify internally calls MEM\_alloc, which may cause a context switch.

## Example

```
/* my notify function prototype */
extern PWRM_NotifyResponse myNotifyFxn(
    PWRM_Event eventType,
    Arg eventArg1,
    Arg eventArg2, Arg clientArg);
...

/* notification handle */
PWRM_NotifyHandle notifyHandle;

/* pointer to return delayed completion fxn */
Fxn delayCompletionFxn;

status = PWRM_registerNotify(PWRM_GOINGTOSLEEP, NULL,
    myNotifyFxn, CLIENT1, &notifyHandle,
    &delayedCompletionFxn);

if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: registration status=%x",
status);
}
```

**pwrnNotifyFxn***Function to be called for power event notification***C Interface**

**Syntax** `status = notifyFxn(eventType, eventArg1, eventArg2, clientArg);`

**Parameters**

|            |            |                               |
|------------|------------|-------------------------------|
| PWRM_Event | eventType; | /* type of power event */     |
| Arg        | eventArg1; | /* event-specific argument */ |
| Arg        | eventArg2; | /* event-specific argument */ |
| Arg        | clientArg; | /* arbitrary argument */      |

**Return Value** `PWRM_NotifyResponse status;` /\* returned status \*/

**Description**

PWRM\_registerNotify registers a function to be called when a specific power event occurs. Clients, which are typically drivers, register notification functions they need to run when a particular power event occurs.

This topic describes the required prototype and behavior of such notification functions. Your application must provide and register these functions. Registered functions are called internally by the PWRM module.

The eventType parameter identifies the type of power event for which the notify function is being called. This parameter has an enumerated type of PWRM\_Event. The values for this parameter are listed in the PWRM\_registerNotify topic.

The eventArg1 and eventArg2 parameters are event-specific arguments. Currently, eventArg1 and eventArg2 are used only for V/F scaling events:

- ☐ **Pending setpoint change for domain**  
(PWRM\_PENDING\_CPU\_SETPOINTCHANGE or PWRM\_PENDING\_PER\_SETPOINTCHANGE). The eventArg1 holds the current setpoint, and eventArg2 holds the pending setpoint.
- ☐ **Done setpoint change for domain**  
(PWRM\_DONE\_CPU\_SETPOINTCHANGE or PWRM\_DONE\_PER\_SETPOINTCHANGE). The eventArg1 holds the previous setpoint, and eventArg2 holds the new setpoint.

The clientArg parameter holds the arbitrary argument passed to PWRM\_registerNotify when this function was registered. This argument may allow one notify function to be used by multiple instances of a driver (that is, the clientArg can be used to identify the instance of the driver that is being notified).

The notification function must return one of the following constants as a status value of type PWRM\_NotifyResponse:

| Name               | Usage                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PWRM_NOTIFYDONE    | The client processed the notification function successfully.                                                                                                                                                                                                                   |
| PWRM_NOTIFYNOTDONE | The client must wait for interrupt processing to occur before it can proceed. The client must later call the delayedCompletionFxn specified when this function was registered with PWRM_registerNotify.                                                                        |
| PWRM_NOTIFYERROR   | Notification cannot be processed. Either an internal client error occurred or the client was notified of an event it could not process. When a client returns this error, the caller of the PWRM function that triggered the notification receives a PWRM_EFAIL return status. |

### Constraints and Calling Context

- ❑ The notification function should not call PWRM APIs that trigger a notification event (PWRM\_sleepDSP). If such an API is called, the PWRM\_EBUSY status code is returned.

### Example

```
/* notification function prototype */
PWRM_NotifyResponse myNotifyFxn1(
    PWRM_Event eventType, Arg eventArg1,
    Arg eventArg2, Arg clientArg);

/* ===== myNotifyFxn1 ===== */
PWRM_NotifyResponse myNotifyFxn1(
    PWRM_Event eventType, Arg eventArg1,
    Arg eventArg2, Arg clientArg)
{
    #if VERBOSE
        LOG_printf(TRACE, "client #1 notify function");
        LOG_printf(TRACE, "eventType=0x%x", eventType);
        LOG_printf(TRACE, "eventArg1=%p, eventArg2=%p",
            eventArg1, eventArg2);
        LOG_printf(TRACE, "clientArg=%p", clientArg);
    #endif
    return(PWRM_NOTIFYDONE);    /* notify complete */
}
```

PWRM\_releaseDependency

Release a dependency that was previously declared

C Interface

|              |                                              |             |                       |
|--------------|----------------------------------------------|-------------|-----------------------|
| Syntax       | status = PWRM_releaseDependency(resourceID); |             |                       |
| Parameters   | Uns                                          | resourceID; | /* resource ID */     |
| Return Value | PWRM_Status                                  | status;     | /* returned status */ |
| Reentrant    | yes                                          |             |                       |

**Description** This function is the companion to PWRM\_setDependency. It releases a resource dependency that was previously set.



Resource IDs are device-specific. They are defined in a PWRM\_Resource enumeration in a device-specific header file. For example, see pwr6748.h for the TMS320C6748 device.

PWRM\_ETOOMANYCALLS is returned if you call PWRM\_releaseDependency when there are no dependencies currently declared for the specified resource (either because all have been released or because none were set).

PWRM\_releaseDependency returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                                |
|--------------------|----------------------------------------------------------------------|
| PWRM_SOK           | The operation succeeded, and dependency has been released.           |
| PWRM_EFAIL         | The operation failed while attempting to release the resource.       |
| PWRM_EINVALIDVALUE | The operation failed because PWRM cannot control this resource.      |
| PWRM_ENOTSUPPORTED | The operation failed because resource tracking is not enabled.       |
| PWRM_EOUTOFRANGE   | The specified resourceID is outside the range of valid resource IDs. |
| PWRM_ETOOMANYCALLS | A dependency was not previously set and was therefore not released.  |

Example

```
#include <pwr.h>
#include <pwr6748.h>

...
/* Release dependency upon UART #1 */
PWRM_releaseDependency(PWRM_RSRC_UART_1);
```

## **PWRM\_resetCPULoadHistory** *Clear the CPU load history buffered by PWRM*

### **C Interface**

**Syntax**                      `status = PWRM_resetCPULoadHistory(sync);`

**Parameters**                `Bool                sync                /* flag for action when slot finalizes */`

**Return Value**              `PWRM_Status status;                /* returned status */`

**Reentrant**                      `yes`

### **Description**



PWRM\_resetCPULoadHistory is used to clear any accumulated CPU load history maintained by PWRM. All of the history slots maintained in PWRM's internal ring buffer are reset to their "empty" state. (See the \_PWRM\_resetLoad structure in the PWRM\_getCPULoad description.)

Calling PWRM\_resetCPULoadHistory does not affect the periodic finalization of history slots by PWRM. The currently accumulating slot finalizes "on schedule" at the same time, just as if PWRM\_resetCPULoadHistory was not called. The sync flag determines what is to be done when the currently accumulating slot completes:

- ❑ If sync is TRUE, then the new load accumulated from the time PWRM\_resetCPULoadHistory is called until the current time is discarded, and no new history data is available from PWRM until the next slot finalizes.
- ❑ If sync is FALSE, then the new load accumulated from the time PWRM\_resetCPULoadHistory is called until the current time is stored in the first history slot. This slot is shorter in duration than a "normal" slot (which has the duration statically configured for PWRM). This shorter duration is indicated in the totalCycles field of this slot's PWRM\_CPULoadInfo structure.

PWRM\_resetCPULoadHistory returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                                                |
|--------------------|--------------------------------------------------------------------------------------|
| PWRM_SOK           | The operation succeeded.                                                             |
| PWRM_EINITFAILURE  | The operation failed because there was a failure during load monitor initialization. |
| PWRM_ENOTSUPPORTED | The operation failed because load monitoring is not enabled.                         |

### **Example**

```
status = PWRM_resetCPULoadHistory(TRUE);
if (status != PWRM_SOK) {
    LOG_printf(TRACE, "Error: status = %x", status);
}
```

PWRM\_setDependency

Declare a dependency upon a resource

C Interface

Syntax

Parameters

Return Value

Reentrant

status = PWRM\_setDependency(resourceID);

UnsresourceID; /\* resource ID \*/

PWRM\_Status status; /\* returned status \*/

yes

Description

This function sets a dependency on a resource. It is the companion to PWRM\_releaseDependency.



Resource IDs are device-specific. They are defined in a PWRM\_Resource enumeration in a device-specific header file. For example, see pwr6748.h for the TMS320C6748 device.

PWRM\_setDependency returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                                |
|--------------------|----------------------------------------------------------------------|
| PWRM_SOK           | The operation succeeded, and dependency has been set.                |
| PWRM_EFAIL         | The operation failed while attempting to release the resource.       |
| PWRM_EINVALIDVALUE | The operation failed because PWRM cannot control this resource.      |
| PWRM_ENOTSUPPORTED | The operation failed because resource tracking is not enabled.       |
| PWRM_EOUTOFRANGE   | The specified resourceID is outside the range of valid resource IDs. |

Example

```
#include <pwr6748.h>
#include <pwr6748.h>
...
/* Declare a driver dependency upon UART #1 */
PWRM_setDependency(PWRM_RSRC_UART_1);
```



**PWRM\_signalEvent***Signal a power event to registered notification clients***C Interface**

**Syntax** `status = PWRM_signalEvent(eventType, eventArg1, eventArg2, notifyTimeout);`

**Parameters**

|            |                |                                          |
|------------|----------------|------------------------------------------|
| PWRM_Event | eventType;     | /* the power event to be signaled */     |
| Arg        | eventArg1;     | /* event argument #1 */                  |
| Arg        | eventArg2;     | /* event argument #2 */                  |
| Uns        | notifyTimeout; | /* max ticks to wait for notification */ |

**Return Value** `PWRM_Status status;` /\* returned status \*/

**Reentrant** yes

**Description**

PWRM\_signalEvent provides a mechanism for an application to signal power events. PWRM\_signalEvent only signals the event occurrence; it does not implement the actual processing associated with the power event.

For example, on a platform where V/F scaling is accomplished on a separate processor, PWRM\_signalEvent can be called by the DSP application before the scaling to notify registered clients on this processor of the pending setpoint change event. Similarly, after the scaling, PWRM\_signalEvent can be called to signal the done setpoint change event. Here PWRM is not orchestrating the change to the V/F setpoint, but its registration, notification, and signaling capabilities allow synchronization between scaling by the other processor and the affected software on this processor.

The parameters eventArg1 and eventArg2 should correspond to the type of event being signaled. (See the pwrM\_notifyFxn description for more information on event arguments.) For example, for a pending setpoint change event, the current and pending setpoints should be specified as eventArg1 and eventArg2.

The notifyTimeout parameter is the maximum amount of time (in system clock ticks) to wait for all registered notification functions (set by PWRM\_registerNotify) to respond to a delayed completion, before declaring failure and returning PWRM\_ETIMEOUT.

PWRM\_signalEvent is intended only for signaling power events that are outside the full control of the PWRM implementation on the current processor. It can be used for V/F scaling notification as described above when PWRM does not implement the actual scaling. For events where PWRM does orchestrate the processing, for example, on platforms

where PWRM performs the actual V/F scaling, or activates a processor sleep mode, PWRM handles the notifications automatically, and PWRM\_signalEvent must not be used for these events. In other words, PWRM\_signalEvent only validates that the indicated eventType is within the range of valid events for the platform; it does not validate whether it makes sense for the application to trigger the signaling of the indicated event, and it does not implement the actual power transition indicated by the event.

PWRM\_signalEvent returns one of the following constants as a status value of type PWRM\_Status:

| Name              | Usage                                                                                                |
|-------------------|------------------------------------------------------------------------------------------------------|
| PWRM_SOK          | The operation succeeded.                                                                             |
| PWRM_EFAIL        | A notification failure occurred.                                                                     |
| PWRM_EINVALDEVENT | The operation failed because eventType is invalid.                                                   |
| PWRM_ETIMEOUT     | A registered notification client did not complete its processing within the specified notifyTimeout. |

Constraints and  
Calling Context

- ❑ PWRM\_signalEvent can be called from a HWI or SWI only if notifyTimeout is 0.

Example

```
status = PWRM_signalEvent(  
    PWRM_DONE_CPU_SETPOINTCHANGE,  
    previousSetpoint, newSetpoint, 0);  
if (status != PWRM_SOK) {  
    LOG_printf(TRACE, "Error: status = %x", status);  
}
```

**PWRM\_sleepDSP***Transition the DSP to a new sleep state***C Interface****Syntax**

```
status = PWRM_sleepDSP(sleepCode, sleepArg, notifyTimeout);
```

**Parameters**

```
Uns          sleepCode;    /* new sleep state */
LgUns        sleepArg;     /* sleepCode-specific argument */
Uns          notifyTimeout; /* maximum time to wait for notification */
```

**Return Value**

```
PWRM_Status status;        /* returned status */
```

**Reentrant**

yes

**Description**

PWRM\_sleepDSP transitions the DSP to a new sleep state.



The sleepCode parameter indicates the new sleep state for the DSP.

The sleepCode parameter indicates the new sleep state for the DSP. The sleep states supported by PWRM usually vary by device. (See the DSP/BIOS release notes to determine which sleep states are available for your device.) For example, the following constants may be used to activate sleep states on the TMS320C6748:

| Name           | Usage                                                                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PWRM_STANDBY   | The GEM is put into a power-saving standby mode. Its clock is turned off at the GEM boundary. This mode has a low latency for wakeup.                                  |
| PWRM_SLEEP     | In addition to putting the GEM into standby, the core voltage is reduced, and the PLLs are slowed down or bypassed.                                                    |
| PWRM_DEEPSLEEP | In addition to the actions for PWRM_SLEEP, the GEM clock is gated up-stream at the power sleep controller, memories are put into retention, and PLLs are powered down. |

The sleepArg parameter is a sleepCode-specific argument. For example, for the TMS320C6748, this parameter is not used for PWRM\_STANDBY, but is used for PWRM\_SLEEP and PWRM\_DEEPSLEEP.

For PWRM\_SLEEP, the sleepArg can be used to override PWRM's default behavior. If sleepArg is NULL, then PWRM reduces the core voltage to 1.0 volts during sleep, and bypasses both PLLs. If sleepArg is non-NULL, it is interpreted as a pointer to a structure of the following type:

```
typedef struct PWRM_SleepOverride {  
    Uns sleepVoltage; /* sleep voltage in millivolts */  
    Uns bypassedPLLs; /* PLLs to bypass during sleep */  
} PWRM_SleepOverride;
```

The sleepVoltage is in millivolts, and must correspond to a voltage level defined by the scaling configuration library. The bypassedPLLs value is treated as a bitmask to indicate which PLLs get bypassed during sleep: PWRM\_PLL0 and/or PWRM\_PLL1. This override mechanism can be used to ensure operation within the TMS320C6748 device constraints. For example, the minimum supported voltage when DDR2 is used is 1.1 volts, and no DDR2 accesses can be made when PLL1 is bypassed. These constraints could be violated in some use cases, for example, when DDR2 memory is used, and there is a possibility that the wakeup interrupt service routine triggers some off-chip code or data accesses. The override mechanism allows the sleep voltage to be increased to 1.1 volts and PLL1 to not be bypassed during sleep.

For PWRM\_DEEPSLEEP, sleepArg defines the DEEPSLEEP signal source that PWRM should configure for wakeup. This must be either PWRM\_RTC\_ALARM or PWRM\_EXTERNAL.

A successful call to PWRM\_sleepDSP returns when the DSP has awoken from the specified sleep state.

The notifyTimeout parameter is the maximum amount of time (in system clock ticks) to wait for registered notification functions (set by PWRM\_registerNotify) to respond to a delayed completion, before declaring failure and returning PWRM\_ETIMEOUT. If the notifyTimeout parameter is zero, then all notification functions must return PWRM\_NOTIFYDONE-they cannot request a delayed completion. If a notification function does not return, the system will hang. The notifyTimeout is not used to abandon a notification function; rather it indicates the amount of time PWRM\_sleepDSP waits for all delayed completion requests to complete. The wait-loop is entered after all notification functions have been invoked.

PWRM\_sleepDSP returns one of the following constants as a status value of type PWRM\_Status:

| Name                 | Usage                                                                                                                     |
|----------------------|---------------------------------------------------------------------------------------------------------------------------|
| PWRM_SOK             | A successful sleep and wake occurred.                                                                                     |
| PWRM_EFAIL           | A general failure occurred. Could not sleep the DSP.                                                                      |
| PWRM_ENOTIMPLEMENTED | The requested sleep mode is not implemented on this platform.                                                             |
| PWRM_ETIMEOUT        | A registered notification client did not complete its (delayed completion) processing within the specified notifyTimeout. |
| PWRM_EBUSY           | The requested operation cannot be performed at this time; PWRM is busy processing a previous request.                     |

Due to the critical system nature of sleep commands, clients that register for sleep notification should make every effort to respond immediately to the sleep event.

The application should treat return values of PWRM\_ETIMEOUT or PWRM\_EFAIL as critical system failures. These values indicate the notification client is unresponsive, and the system is in an unknown state.

### Constraints and Calling Context

- ❑ PWRM\_sleepDSP cannot be called from an HWI.
- ❑ This API cannot be called from a program's main() function.
- ❑ PWRM\_sleepDSP can be called from a SWI only if notifyTimeout is 0.
- ❑ For a sleepCode of PWRM\_SLEEP, no DDR memory accesses (code, data, or stack) can occur while the PLL1 is bypassed, otherwise DDR corruption can occur.
- ❑ For a sleepCode of PWRM\_SLEEP, if DDR2 memories are used, then no DDR2 accesses can occur with a sleep voltage of 1.0 volts.
- ❑ Using a sleepCode of PWRM\_DEEPSLEEP must be invoked from an on-chip context; no code, data, or stack access can occur during PWRM\_sleepDSP, otherwise DDR corruption may occur.

### Example

```
#define TIMEOUT    10    /* timeout after 10 ticks */

LOG_printf	TRACE, "Putting DSP to sleep...\n");
status = PWRM_sleepDSP(PWRM_SLEEP, NULL, TIMEOUT);
LOG_printf	TRACE, "DSP awake from sleep";
LOG_printf	TRACE, "Returned sleep status 0x%x",
            status);
```

PWRM\_startCPULoadMonitoring

Restart CPU load monitoring

C Interface

|              |                                                                                                           |
|--------------|-----------------------------------------------------------------------------------------------------------|
| Syntax       | status = PWRM_startCPULoadMonitoring();                                                                   |
| Parameters   | none                                                                                                      |
| Return Value | PWRM_Status status; /* returned status */                                                                 |
| Reentrant    | yes                                                                                                       |
| Description  | This function restarts the collection of CPU load information for the purpose of monitoring the CPU load. |



When load monitoring is enabled in the application configuration for PWRM, the monitoring starts automatically as part of DSP/BIOS initialization. Calling this API is only necessary if the application has previously called PWRM\_stopCPULoadMonitoring.

When CPU load monitoring is started, an internal call to PWRM\_resetCPULoadHistory(TRUE) is made.

PWRM\_startCPULoadMonitoring returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                                                |
|--------------------|--------------------------------------------------------------------------------------|
| PWRM_SOK           | The operation succeeded.                                                             |
| PWRM_ENOTSUPPORTED | The operation failed because load monitoring is not enabled.                         |
| PWRM_EINITFAILURE  | The operation failed because there was a failure during load monitor initialization. |

Example

status = PWRM\_startCPULoadMonitoring();

**PWRM\_stopCPULoadMonitoring***Stop CPU load monitoring***C Interface**

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = PWRM_stopCPULoadMonitoring();</code>           |
| <b>Parameters</b>   | none                                                          |
| <b>Return Value</b> | <code>PWRM_Status status;</code> <i>/* returned status */</i> |
| <b>Reentrant</b>    | yes                                                           |

**Description**

This function stops the collection of CPU load information for the purpose of monitoring the CPU load. While stopped, no calls are made to the configured slot finalization hook function.

PWRM\_stopCPULoadMonitoring returns one of the following constants as a status value of type PWRM\_Status:

| <b>Name</b> | <b>Usage</b>             |
|-------------|--------------------------|
| PWRM_SOK    | The operation succeeded. |

**Example**

```
status = PWRM_stopCPULoadMonitoring();
```

PWRM\_unregisterConstraint

Unregister a constraint from PWRM

C Interface

|              |                                                       |                   |                         |
|--------------|-------------------------------------------------------|-------------------|-------------------------|
| Syntax       | status = PWRM_unregisterConstraint(constraintHandle); |                   |                         |
| Parameters   | PWRM_ConstraintHandle                                 | constraintHandle; | /* constraint handle */ |
| Return Value | PWRM_Status                                           | status;           | /* returned status */   |
| Reentrant    | yes                                                   |                   |                         |

Description

PWRM\_unregisterConstraint unregisters a constraint that was registered by PWRM\_registerConstraint. For example, when a device driver is closed, any constraints it registered with PWRM must be unregistered, otherwise PWRM may be left unnecessarily restricted from activating power savings.



The constraintHandle parameter is the parameter that was provided by PWRM\_registerConstraint when the constraint was registered.

PWRM\_unregisterConstraint returns one of the following constants as a status value of type PWRM\_Status:

| Name               | Usage                                                     |
|--------------------|-----------------------------------------------------------|
| PWRM_SOK           | The constraint was successfully unregistered.             |
| PWRM_EFAIL         | The operation failed due to a memory free failure.        |
| PWRM_EINVALIDEVENT | The operation failed because constraintHandle is invalid. |

Constraints and Calling Context

❑ This API cannot be called from a program's main() function.

Example

```
PWRM_ConstraintHandle constraintHandle;

status = PWRM_registerConstraint(
    PWRM_DISALLOWEDSLEEPMODE_MASK,
    (PWRM_SLEEP | PWRM_DEEPSLEEP),
    &constraintHandle);
...
PWRM_unregisterConstraint(constraintHandle);
```



## C Interface

|                     |             |         |                       |
|---------------------|-------------|---------|-----------------------|
| <b>Return Value</b> | PWRM_Status | status; | /* returned status */ |
|---------------------|-------------|---------|-----------------------|

### Description



PWRM\_unregisterNotify returns one of the following constants as a status value of type PWRM\_Status:

| Name                | Usage                                              |
|---------------------|----------------------------------------------------|
| PWRM_SOK            | The function was successfully unregistered.        |
| PWRM_EFAIL          | The operation failed due to a memory free failure. |
| PWRM_EINVALIDHANDLE | Operation failed because notifyHandle is invalid.  |

- ❑ This API cannot be called from a program's `main()` function.

### Example

```
PWRM_NotifyHandle notifyHandle;

PWRM_registerNotify(PWRM_GOINGTOSLEEP, NULL,
    (Fxn)myNotifyFxn, (Arg)0x1111, &notifyHandle,
    (Fxn *) &delayFxn);
...
PWRM_unregisterNotify(notifyHandle);
```

2.24 QUE Module

Functions

The QUE module is the atomic queue manager.

- ❑ 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 */
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the QUE Manager Properties and QUE Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

Module Configuration Parameters

| Name      | Type      | Default           |
|-----------|-----------|-------------------|
| OBJMEMSEG | Reference | prog.get("IDRAM") |

## Instance Configuration Parameters

| Name    | Type   | Default               |
|---------|--------|-----------------------|
| comment | String | "<add comments here>" |

### 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 a structure. The `QUE_Elem` item is used as an internal pointer.

For example, the `DEV_Frame` structure, which is used by the SIO Module and DEV Module to enqueue and dequeue I/O buffers, contains a field of type `QUE_Elem`:

```
struct DEV_Frame {      /* frame object */
    QUE_Elem  link;      /* must be first field! */
    Ptr       addr;      /* buffer address */
    size_t    size;      /* buffer size */
    Arg       misc;      /* reserved for driver */
    Arg       arg;       /* user argument */
    Uns       cmd;       /* mini-driver command */
    Int       status;    /* status of command */
} DEV_Frame;
```

Many QUE module functions either are passed or return a pointer to an element having the structure defined for QUE elements.

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.

### QUE Manager Properties

The following global property can be set for the QUE module in the QUE Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.QUE.OBJMEMSEG = prog.get("myMEM");`

## QUE Object Properties

To create a QUE object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myQue = bios.QUE.create("myQue");
```

The following property can be set for a QUE object in the PRD Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name: comment

Type: String

Example: `myQue.comment = "my QUE";`

## QUE\_create

*Create an empty queue*

### C Interface

**Syntax**                      `queue = QUE_create(attrs);`

**Parameters**                `QUE_Attrs    *attrs;        /* pointer to queue attributes */`

**Return Value**              `QUE_Handle queue;        /* handle for new queue object */`

### Description

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).

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.

#### 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.

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 2-240.

### Constraints and Calling Context

- ❑ QUE\_create cannot be called from a SWI or HWI.
- ❑ You can reduce the size of your application program by creating objects with the Tconf 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****Syntax**`QUE_delete(queue);`**Parameters**`QUE_Handle queue;    /* queue handle */`**Return Value**`Void`**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 a 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 Tconf, SYS\_error is called.

**See Also**

QUE\_create  
QUE\_empty

**QUE\_dequeue***Remove from front of queue (non-atomically)***C Interface**

|                     |                                                                                |
|---------------------|--------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>elem = QUE_dequeue(queue);</code>                                        |
| <b>Parameters</b>   | <code>QUE_Handle queue;</code> <i>/* queue object handle */</i>                |
| <b>Return Value</b> | <code>Ptr            elem;</code> <i>/* pointer to former first element */</i> |

**Description**

QUE\_dequeue removes the element from the front of queue and returns elem.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

Calling QUE\_dequeue with an empty queue returns the queue itself. However, QUE\_dequeue is non-atomic. Therefore, the method described for QUE\_get of checking to see if a queue is empty and returning the first element otherwise is non-atomic.

---

**Note:**

You should use QUE\_get instead of QUE\_dequeue if multiple threads share a queue. QUE\_get runs atomically and is never interrupted; QUE\_dequeue performs the same action but runs non-atomically. You can use QUE\_dequeue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue. An HWI or task that preempts QUE\_dequeue and operates on the same queue can corrupt the data structure.

QUE\_dequeue is somewhat faster than QUE\_get, but 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\_get

## QUE\_empty

*Test for an empty queue*

### C Interface

**Syntax**                      `empty = QUE_empty(queue);`

**Parameters**                `QUE_Handle queue;    /* queue object handle */`

**Return Value**              `Bool                  empty;    /* TRUE if queue is empty */`

**Description**                `QUE_empty` returns TRUE if there are no elements in queue, and FALSE otherwise.

**See Also**                    `QUE_get`



**QUE\_enqueue***Insert at end of queue (non-atomically)***C Interface****Syntax**`QUE_enqueue(queue, elem);`**Parameters**`QUE_Handle queue; /* queue object handle */  
Ptr elem; /* pointer to queue element */`**Return Value**`Void`**Description**

QUE\_enqueue inserts elem at the end of queue.

The elem parameter must be a pointer to an element to be placed in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

---

**Note:**

Use QUE\_put instead of QUE\_enqueue if multiple threads share a queue. QUE\_put is never interrupted; QUE\_enqueue performs the same action but runs non-atomically. You can use QUE\_enqueue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE\_enqueue is somewhat faster than QUE\_put, but 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****Syntax**

```
elem = QUE_get(queue);
```

**Parameters**

```
QUE_Handle queue;    /* queue object handle */
```

**Return Value**

```
Void                *elem;    /* pointer to former first element */
```

**Description**

QUE\_get removes the element from the front of queue and returns elem.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

Since QUE\_get manipulates the queue with interrupts disabled, the 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 `
```

---

**Note:**

Use QUE\_get instead of QUE\_dequeue if multiple threads share a queue. QUE\_get is never interrupted; QUE\_dequeue performs the same action but runs non-atomically. You can use QUE\_dequeue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE\_dequeue is somewhat faster than QUE\_get, but 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\_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 \*/

**Description**

QUE\_head returns a pointer to the element at the front of queue. The element is not removed from the queue.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

Calling QUE\_head with an empty queue returns the queue itself.

**See Also**

QUE\_create  
QUE\_empty  
QUE\_put

**QUE\_insert**

*Insert in middle of queue (non-atomically)*

**C Interface**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | QUE_insert(qelem, elem);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Parameters</b>   | <div>Ptr            qelem;    /* element already in queue */<br/>Ptr            elem;     /* element to be inserted in queue */</div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Return Value</b> | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Description</b>  | <p>QUE_insert inserts elem in the queue in front of qelem.</p> <p>The qelem parameter is a pointer to an existing element of the QUE. The elem parameter is a pointer to an element to be placed in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.</p> <hr/> <p><b>Note:</b></p> <p>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).</p> <hr/> |
| <b>See Also</b>     | <div>QUE_head<br/>QUE_next<br/>QUE_prev<br/>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                                               |

**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.

**See Also**

QUE\_create  
QUE\_delete  
QUE\_empty

**QUE\_next**

*Return next element in queue (non-atomically)*

**C Interface**

|                     |                                                                   |
|---------------------|-------------------------------------------------------------------|
| <b>Syntax</b>       | <code>elem = QUE_next(qelem);</code>                              |
| <b>Parameters</b>   | <code>Ptr            qelem;    /* element in queue */</code>      |
| <b>Return Value</b> | <code>Ptr            elem;     /* next element in queue */</code> |

**Description**            `QUE_next` returns `elem` which points to the element in the queue after `qelem`.

The `qelem` parameter is a pointer to an existing element of the QUE. The return value, `elem`, is a pointer to the next element in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type `QUE_Elem` and is used as an internal pointer.

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.

---

**Note:**

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`).

---

**See Also**

`QUE_get`  
`QUE_insert`  
`QUE_prev`  
`QUE_remove`

## QUE\_prev

*Return previous element in queue (non-atomically)*

### C Interface

**Syntax** `elem = QUE_prev(qelem);`

**Parameters** `Ptr qelem; /* element in queue */`

**Return Value** `Ptr elem; /* previous element in queue */`

### Description

QUE\_prev returns elem which points to the element in the queue before qelem.

The qelem parameter is a pointer to an existing element of the QUE. The return value, elem, is a pointer to the previous element in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

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****Syntax**`QUE_put(queue, elem);`**Parameters**`QUE_Handle queue; /* queue object handle */  
Void *elem; /* pointer to new queue element */`**Return Value**`Void`**Description**

QUE\_put puts elem at the end of queue.

The elem parameter is a pointer to an element to be placed at the end of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

Since QUE\_put manipulates queues with interrupts disabled, queues can be shared by multiple tasks, or by tasks and SWIs or HWIs.

---

**Note:**

Use QUE\_put instead of QUE\_enqueue if multiple threads share a queue. QUE\_put is never interrupted; QUE\_enqueue performs the same action but runs non-atomically. You can use QUE\_enqueue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE\_enqueue is somewhat faster than QUE\_put, but 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_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                                            |

**Description**

QUE\_remove removes qelem from the queue.

The qelem parameter is a pointer to an existing element to be removed from the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE\_Elem and is used as an internal pointer.

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(qelem);
}
/* 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, 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

## 2.25 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

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the RTDX Manager Properties and RTDX Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

| Name          | Type       | Default (Enum Options)         |
|---------------|------------|--------------------------------|
| ENABLERTDX    | Bool       | true                           |
| MODE          | EnumString | "JTAG" ("HSRTDX", "Simulator") |
| RTDXDATASEG   | Reference  | prog.get("IDRAM")              |
| BUFSIZE       | Int16      | 1032                           |
| INTERRUPTMASK | Int16      | 0x00000000                     |

#### Instance Configuration Parameters

| Name        | Type       | Default (Enum Options) |
|-------------|------------|------------------------|
| comment     | String     | "<add comments here>"  |
| channelMode | EnumString | "output" ("input")     |

### 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.

The RTDX assembly interface, *rtdx.i*, is a macro interface file that can be used to interface to RTDX at the assembly level.

- ☐ **Enable Real-Time Data Exchange (RTDX).** This property should be set to true if you want to link RTDX support into your application.

Type: Bool

- ❑ **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 use a simulator. The HS-RTDX emulation technology is also available. If this property is set incorrectly, a message says “RTDX target application does not match emulation protocol” when you load the program.

Type: EnumString

Example: `bios.RTDX.MODE = "JTAG";`

- ❑ **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.

Type: Reference

```
Example: bios.RTDX.RTDXDATASEG =
         prog.get("myMEM");
```

- ❑ **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.

Type: Int16

Example: `bios.RTDX.BUFSIZE = 1032;`

- ❑ **RTDX Interrupt Mask.** This mask interrupts to be temporarily disabled inside critical RTDX sections. The default value of zero (0) disables all interrupts within critical RTDX sections. Such sections are short (usually <100 cycles). Disabling interrupts also temporarily disables other RTDX clients and prevents other RTDX function calls.

You should allow all interrupts to be disabled inside critical RTDX sections if your application makes any RTDX calls from SWI or TSK threads. If your application does not make RTDX calls from SWI or TSK threads, you may modify bits in this mask to enable specific high-priority interrupts. See the RTDX documentation for details.

Tconf Name: INTERRUPTMASK Type: Int16

Example: `bios.RTDX.INTERRUPTMASK = 0x00000000;`

## RTDX Object Properties

To create an RTDX object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myRtdx = bios.RTDX.create("myRtdx");
```

The following properties can be set for an RTDX object in the RTDX Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this RTDX object.

Tconf Name: comment

Type: String

Example: `myRtdx.comment = "my RTDX";`

- ❑ **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.

Tconf Name: channelMode

Type: EnumString

Options: "input", "output"

Example: `myRtdx.channelMode = "output";`

## Examples

The `rtdx.xls` example is in the `TI_DIR\examples\hostapps\rtdx` folder. 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**

|                                        |                                                                                                                                                                                                                                                                                                            |                                                                             |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <b>Syntax</b>                          | int RTDX_channelBusy( RTDX_inputChannel *pichan );                                                                                                                                                                                                                                                         |                                                                             |
| <b>Parameters</b>                      | pichan                                                                                                                                                                                                                                                                                                     | /* Identifier for the input data channel */                                 |
| <b>Return Value</b>                    | int                                                                                                                                                                                                                                                                                                        | /* Status: 0 = Channel is not busy. */<br>/* non-zero = Channel is busy. */ |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                                                                                                        |                                                                             |
| <b>Description</b>                     | 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. |                                                                             |
| <b>Constraints and Calling Context</b> | ❑ RTDX_channelBusy cannot be called by an HWI function.                                                                                                                                                                                                                                                    |                                                                             |
| <b>See Also</b>                        | RTDX_readNB                                                                                                                                                                                                                                                                                                |                                                                             |

**RTDX\_CreateInputChannel***Declare input channel structure***C Interface**

|                     |                                                              |
|---------------------|--------------------------------------------------------------|
| <b>Syntax</b>       | RTDX_CreateInputChannel( ichan );                            |
| <b>Parameters</b>   | ichan                      /* Label for the input channel */ |
| <b>Return Value</b> | none                                                         |

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

|                    |                                                                            |
|--------------------|----------------------------------------------------------------------------|
| <b>Description</b> | This macro declares and initializes to 0, the RTDX data channel for input. |
|--------------------|----------------------------------------------------------------------------|

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.

A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.

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.

**Constraints and Calling Context**

- ❑ RTDX\_CreateInputChannel cannot be called by an HWI function.

|                 |                          |
|-----------------|--------------------------|
| <b>See Also</b> | RTDX_CreateOutputChannel |
|-----------------|--------------------------|

RTDX\_CreateOutputChannel

Declare output channel structure

C Interface

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax                          | RTDX_CreateOutputChannel( ochan );                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parameters                      | ochan                      /* Label for the output channel */                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Return Value                    | none                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Reentrant                       | no                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description                     | <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> |
| Constraints and Calling Context | <div><div>❑</div>RTDX_CreateOutputChannel cannot be called by an HWI function.</div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| See Also                        | RTDX_CreateInputChannel                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |



**RTDX\_disableInput***Disable an input data channel***C Interface**

|                                        |                                                                                                          |
|----------------------------------------|----------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>void RTDX_disableInput( RTDX_inputChannel *ichan );</code>                                         |
| <b>Parameters</b>                      | <code>ichan</code> <i>/* Identifier for the input data channel */</i>                                    |
| <b>Return Value</b>                    | <code>void</code>                                                                                        |
| <b>Reentrant</b>                       | yes                                                                                                      |
| <b>Description</b>                     | A call to a disable function causes the specified input channel to be disabled.                          |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ RTDX_disableInput cannot be called by an HWI function.</li></ul> |
| <b>See Also</b>                        | RTDX_disableOutput<br>RTDX_enableInput<br>RTDX_read                                                      |

**RTDX\_disableOutput** *Disable an output data channel***C Interface**

**Syntax** void RTDX\_disableOutput( RTDX\_outputChannel \*ochan );

**Parameters** ochan /\* Identifier for an output data channel \*/

**Return Value** void

**Reentrant** yes

**Description** A call to a disable function causes the specified data channel to be disabled.

**Constraints and Calling Context** ☐ RTDX\_disableOutput cannot be called by an HWI function.

**See Also** RTDX\_disableInput  
RTDX\_enableOutput  
RTDX\_read

**RTDX\_enableInput** *Enable an input data channel***C Interface**

|                                        |                                                                                                                                             |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | void RTDX_enableInput( RTDX_inputChannel *ichan );                                                                                          |
| <b>Parameters</b>                      | ochan                    /* Identifier for an output data channel */<br>ichan                   /* Identifier for the input data channel */ |
| <b>Return Value</b>                    | void                                                                                                                                        |
| <b>Reentrant</b>                       | yes                                                                                                                                         |
| <b>Description</b>                     | A call to an enable function causes the specified data channel to be enabled.                                                               |
| <b>Constraints and Calling Context</b> | ❑ RTDX_enableInput cannot be called by an HWI function.                                                                                     |
| <b>See Also</b>                        | RTDX_disableInput<br>RTDX_enableOutput<br>RTDX_read                                                                                         |

**RTDX\_enableOutput**    *Enable an output data channel***C Interface**

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| <b>Syntax</b>       | void RTDX_enableOutput( RTDX_outputChannel *ochan );                   |
| <b>Parameters</b>   | ochan                      /* Identifier for an output data channel */ |
| <b>Return Value</b> | void                                                                   |

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

|                    |                                                                               |
|--------------------|-------------------------------------------------------------------------------|
| <b>Description</b> | A call to an enable function causes the specified data channel to be enabled. |
|--------------------|-------------------------------------------------------------------------------|

|                                        |                                                                                                          |
|----------------------------------------|----------------------------------------------------------------------------------------------------------|
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ RTDX_enableOutput cannot be called by an HWI function.</li></ul> |
|----------------------------------------|----------------------------------------------------------------------------------------------------------|

|                 |                                                      |
|-----------------|------------------------------------------------------|
| <b>See Also</b> | RTDX_disableOutput<br>RTDX_enableInput<br>RTDX_write |
|-----------------|------------------------------------------------------|

**RTDX\_isInputEnabled***Return status of the input data channel***C Interface**

|                     |                               |                                        |
|---------------------|-------------------------------|----------------------------------------|
| <b>Syntax</b>       | RTDX_isInputEnabled( ichan ); |                                        |
| <b>Parameter</b>    | ichan                         | /* Identifier for an input channel. */ |
| <b>Return Value</b> | 0                             | /* Not enabled. */                     |
|                     | non-zero                      | /* Enabled. */                         |

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

|                    |                                                                                                                                                                                                                  |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Description</b> | 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. |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                        |                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------|
| <b>Constraints and Calling Context</b> | <input type="checkbox"/> RTDX_isInputEnabled cannot be called by an HWI function. |
|----------------------------------------|-----------------------------------------------------------------------------------|

|                 |                      |
|-----------------|----------------------|
| <b>See Also</b> | RTDX_isOutputEnabled |
|-----------------|----------------------|

**RTDX\_isOutputEnabled**    *Return status of the output data channel*

**C Interface**

|                                        |                                                                                                                                                                                                                     |                                         |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|
| <b>Syntax</b>                          | RTDX_isOutputEnabled(ohan );                                                                                                                                                                                        |                                         |
| <b>Parameter</b>                       | ochan                                                                                                                                                                                                               | /* Identifier for an output channel. */ |
| <b>Return Value</b>                    | 0                                                                                                                                                                                                                   | /* Not enabled. */                      |
|                                        | non-zero                                                                                                                                                                                                            | /* Enabled. */                          |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                 |                                         |
| <b>Description</b>                     | 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. |                                         |
| <b>Constraints and Calling Context</b> | ❑ RTDX_isOutputEnabled cannot be called by an HWI function.                                                                                                                                                         |                                         |
| <b>See Also</b>                        | RTDX_isInputEnabled                                                                                                                                                                                                 |                                         |

**RTDX\_read***Read from an input channel***C Interface**

**Syntax** `int RTDX_read( RTDX_inputChannel *ichan, void *buffer, int bsize );`

**Parameters**

|                     |                                                                   |
|---------------------|-------------------------------------------------------------------|
| <code>ichan</code>  | <code>/* Identifier for the input data channel */</code>          |
| <code>buffer</code> | <code>/* A pointer to the buffer that receives the data */</code> |
| <code>bsize</code>  | <code>/* The size of the buffer in address units */</code>        |

**Return Value**

|                              |                                                                                                           |
|------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>&gt; 0</code>          | <code>/* The number of address units of data */</code><br><code>/* actually supplied in buffer. */</code> |
| <code>0</code>               | <code>/* Failure. Cannot post read request */</code><br><code>/* because target buffer is full. */</code> |
| <code>RTDX_READ_ERROR</code> | <code>/* Failure. Channel currently busy or</code><br><code>not enabled. */</code>                        |

**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\_readNB**

*Read from input channel without blocking*

**C Interface**

|                     |                                                                       |                                                      |
|---------------------|-----------------------------------------------------------------------|------------------------------------------------------|
| <b>Syntax</b>       | int RTDX_readNB( RTDX_inputChannel *ichan, void *buffer, int bsize ); |                                                      |
| <b>Parameters</b>   | 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 */        |
| <b>Return Value</b> | RTDX_OK                                                               | /* Success.*/                                        |
|                     | 0 (zero)                                                              | /* Failure. The target buffer is full. */            |
|                     | RTDX_READ_ERROR                                                       | /*Channel is currently busy reading. */              |
| <b>Reentrant</b>    | 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**

|                                        |                                                                                                                                                                                                                                                |                                                                           |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| <b>Syntax</b>                          | int RTDX_sizeofInput( RTDX_inputChannel *pichan );                                                                                                                                                                                             |                                                                           |
| <b>Parameters</b>                      | pichan                                                                                                                                                                                                                                         | /* Identifier for the input data channel */                               |
| <b>Return Value</b>                    | int                                                                                                                                                                                                                                            | /* Number of sizeof units of data actually */<br>/* supplied in buffer */ |
| <b>Reentrant</b>                       | yes                                                                                                                                                                                                                                            |                                                                           |
| <b>Description</b>                     | 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). |                                                                           |
| <b>Constraints and Calling Context</b> | ❑ RTDX_sizeofInput cannot be called by an HWI function.                                                                                                                                                                                        |                                                                           |
| <b>See Also</b>                        | RTDX_readNB                                                                                                                                                                                                                                    |                                                                           |

RTDX\_write

Write to an output channel

C Interface

|                                 |                                                                                                                                                                                                                                                                                                                                                                              |                                                   |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| Syntax                          | int RTDX_write( RTDX_outputChannel *ochan, void *buffer, int bsize );                                                                                                                                                                                                                                                                                                        |                                                   |
| Parameters                      | ochan                                                                                                                                                                                                                                                                                                                                                                        | /* Identifier for the output data channel */      |
|                                 | buffer                                                                                                                                                                                                                                                                                                                                                                       | /* A pointer to the buffer containing the data */ |
|                                 | bsize                                                                                                                                                                                                                                                                                                                                                                        | /* The size of the buffer in address units */     |
| Return Value                    | int                                                                                                                                                                                                                                                                                                                                                                          | /* Status: non-zero = Success. 0 = Failure. */    |
| 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                                                                                                                                                                                                                                                                                                                                                                    |                                                   |

## 2.26 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\_new. Initialize a semaphore
- ❑ SEM\_pend. Wait for a counting semaphore
- ❑ SEM\_pendBinary. Wait for a binary semaphore
- ❑ SEM\_post. Signal a counting semaphore
- ❑ SEM\_postBinary. Signal a binary semaphore
- ❑ SEM\_reset. Reset semaphore

### Constants, Types, and Structures

```
typedef struct SEM_Obj  *SEM_Handle;
                        /* handle for semaphore object */

struct SEM_Attrs { /* semaphore attributes */
    String name; /* printable name */
};

SEM_Attrs SEM_ATTRS = { /* default attribute values */
    "", /* name */
};
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SEM Manager Properties and SEM Object Properties topics. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

#### Module Configuration Parameters

| Name      | Type      | Default           |
|-----------|-----------|-------------------|
| OBJMEMSEG | Reference | prog.get("IDRAM") |

#### Instance Configuration Parameters

| Name    | Type   | Default               |
|---------|--------|-----------------------|
| comment | String | "<add comments here>" |
| count   | Int16  | 0                     |

## Description

The SEM module makes available a set of functions that manipulate semaphore objects accessed through handles of type `SEM_Handle`. Semaphores can be used for task synchronization and mutual exclusion.

Semaphores can be counting semaphores or binary semaphores. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

- ❑ **Counting semaphores** keep track of the number of times the semaphore has been posted with `SEM_post`. This is useful, for example, if you have a group of resources that are shared between tasks. Such tasks might call `SEM_pend` to see if a resource is available before using one. `SEM_pend` and `SEM_post` are for use with counting semaphores.
- ❑ **Binary semaphores** can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to `SEM_pendBinary` clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not. `SEM_pendBinary` and `SEM_postBinary` are for use with binary semaphores.

The MBX module uses a counting semaphore internally to manage the count of free (or full) mailbox elements. Another example of a counting semaphore is an ISR that might fill multiple buffers of data for consumption by a task. After filling each buffer, the ISR puts the buffer on a queue and calls `SEM_post`. The task waiting for the data calls `SEM_pend`, which simply decrements the semaphore count and returns or blocks if the count is 0. The semaphore count thus tracks the number of full buffers available for the task. The GIO and SIO modules follow this model and use counting semaphores.

The internal data structures used for binary and counting semaphores are the same; the only change is whether semaphore values are incremented and decremented or simply set to zero and non-zero.

`SEM_pend` and `SEM_pendBinary` are used to wait for a semaphore. The timeout parameter allows the task to wait until a timeout, wait indefinitely, or not wait at all. The return value is used to indicate if the semaphore was signaled successfully.

`SEM_post` and `SEM_postBinary` are used to signal a semaphore. If a task is waiting for the semaphore, `SEM_post/SEM_postBinary` removes the task from the semaphore queue and puts it on the ready queue. If no

## SEM Manager Properties

- ❑ **Object Memory.** The memory segment that contains the SEM objects created with Tconf.

Example: `bios.SEM.OBJMEMSEG = prog.get("myMEM");`

To create a SEM object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

The following properties can be set for a SEM object in the SEM Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- |                     |              |
|---------------------|--------------|
| Tconf Name: comment | Type: String |
|---------------------|--------------|

Example: `mySem.comment = "my SEM";`

- ❑ **Initial semaphore count.** Set this property to the desired initial semaphore count.

Tconf Name: count Type: Int16

Example: `mySem.count = 0;`

SEM\_count

Get current semaphore count

C Interface

|              |                                                                        |
|--------------|------------------------------------------------------------------------|
| Syntax       | count = SEM_count(sem);                                                |
| Parameters   | SEM_Handle sem;      /* semaphore handle */                            |
| Return Value | Int                  count;      /* current semaphore count */         |
| Description  | 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 \*/

**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.

```
struct SEM_Attrs { /* semaphore attributes */
    String name; /* printable name */
};
```

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_Attrs SEM_ATTRS = { /* default attribute values */
    "", /* name */
};
```

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 a SWI or HWI.
- ❑ You can reduce the size of your application by creating objects with Tconf rather than XXX\_create functions.

**See Also**

MEM\_alloc  
 SEM\_delete

**SEM\_delete***Delete a semaphore***C Interface**

|                     |                                                    |
|---------------------|----------------------------------------------------|
| <b>Syntax</b>       | SEM_delete(sem);                                   |
| <b>Parameters</b>   | SEM_Handle sem;      /* semaphore object handle */ |
| <b>Return Value</b> | Void                                               |

**Description** SEM\_delete uses MEM\_free to free the semaphore object referenced by sem.

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.

**Constraints and  
Calling Context**

- ❑ No tasks should be pending on sem when SEM\_delete is called.
- ❑ SEM\_delete cannot be called from a SWI or HWI.
- ❑ 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 Tconf, SYS\_error is called.

**See Also** SEM\_create



**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

**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**

- ☐ count must be greater than or equal to 0
- ☐ no tasks should be pending on the semaphore when SEM\_new is called

**See Also**

QUE\_new

**SEM\_pend***Wait for a semaphore***C Interface**

|                     |                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = SEM_pend(sem, timeout);</code>                                                                                                                    |
| <b>Parameters</b>   | <code>SEM_Handle sem;</code> <i>/* semaphore object handle */</i><br><code>Uns            timeout;</code> <i>/* return after this many system clock ticks */</i> |
| <b>Return Value</b> | <code>Bool           status;</code> <i>/* TRUE if successful, FALSE if timeout */</i>                                                                            |

**Description**

SEM\_pend and SEM\_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. In contrast, SEM\_pendBinary and SEM\_postBinary are for use with binary semaphores, which can have only an available or unavailable state. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

If the semaphore count is greater than zero (available), SEM\_pend decrements the count and returns TRUE. If the semaphore count is zero (unavailable), SEM\_pend suspends execution of the current task until SEM\_post is called or the timeout expires.

If timeout is SYS\_FOREVER, a task stays 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.

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.

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 be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- ☐ SEM\_pend cannot be called from the program's main() function.
- ☐ If you need to call SEM\_pend within a TSK\_disable/TSK\_enable block, you must use a timeout of 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\_pendBinary  
SEM\_post

## SEM\_pendBinary *Wait for a binary semaphore*

### C Interface

|                     |                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>status = SEM_pendBinary(sem, timeout);</code>                                                                                                              |
| <b>Parameters</b>   | <code>SEM_Handle sem;</code> <i>/* semaphore object handle */</i><br><code>Uns            timeout;</code> <i>/* return after this many system clock ticks */</i> |
| <b>Return Value</b> | <code>Bool            status;</code> <i>/* TRUE if successful, FALSE if timeout */</i>                                                                           |

### Description

SEM\_pendBinary and SEM\_postBinary are for use with binary semaphores. These are semaphores that can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to SEM\_pendBinary clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not.

In contrast, SEM\_pend and SEM\_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

If the semaphore count is non-zero (available), SEM\_pendBinary sets the count to zero (unavailable) and returns TRUE.

If the semaphore count is zero (unavailable), SEM\_pendBinary suspends execution of this task until SEM\_post is called or the timeout expires.

If timeout is SYS\_FOREVER, a task remains suspended until SEM\_postBinary is called on this semaphore. If timeout is 0, SEM\_pendBinary returns immediately.

If timeout expires (or timeout is 0) before the semaphore is available, SEM\_pendBinary returns FALSE. Otherwise SEM\_pendBinary returns TRUE.

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.

A task switch occurs when calling SEM\_pendBinary if the semaphore count is 0 and timeout is not zero.

### Constraints and Calling Context

- ❑ This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.

- ❑ This API cannot be called from the program's `main()` function.
- ❑ If you need to call this API within a `TSK_disable/TSK_enable` block, you must use a timeout of 0.
- ❑ This API should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.

**See Also**

`SEM_pend`  
`SEM_postBinary`

**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                                               |

**Description**

SEM\_pend and SEM\_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks.

In contrast, SEM\_pendBinary and SEM\_postBinary are for use with binary semaphores, which can have only an available or unavailable state. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

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, the code sequence calling SEM\_post must be either wrapped within an HWI\_enter/HWI\_exit pair or invoked by the HWI dispatcher.
- ❑ If SEM\_post is called from within a TSK\_disable/TSK\_enable block, the semaphore operation is not processed until TSK\_enable is called.

**See Also**

SEM\_pend  
SEM\_postBinary

**SEM\_postBinary***Signal a binary semaphore***C Interface**

|                     |                                                                   |
|---------------------|-------------------------------------------------------------------|
| <b>Syntax</b>       | <code>SEM_postBinary(sem);</code>                                 |
| <b>Parameters</b>   | <code>SEM_Handle sem;</code> <i>/* semaphore object handle */</i> |
| <b>Return Value</b> | <code>Void</code>                                                 |

**Description**

`SEM_pendBinary` and `SEM_postBinary` are for use with binary semaphores. These are semaphores that can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to `SEM_pendBinary` clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not.

In contrast, `SEM_pend` and `SEM_post` are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

`SEM_postBinary` readies the first task in the list if one or more tasks are waiting. `SEM_postBinary` sets the semaphore count to non-zero (available) if no tasks are waiting.

A task switch occurs when calling `SEM_postBinary` if a higher priority task is made ready to run.

**Constraints and Calling Context**

- ❑ When called within an HWI, the code sequence calling this API must be either wrapped within an `HWI_enter/HWI_exit` pair or invoked by the HWI dispatcher.
- ❑ If this API is called from within a `TSK_disable/TSK_enable` block, the semaphore operation is not processed until `TSK_enable` is called.

**See Also**

`SEM_post`  
`SEM_pendBinary`

**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`**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`

## 2.27 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\_ready. Determine if device is ready
- ❑ SIO\_reclaim. Request a buffer back from a stream
- ❑ SIO\_reclaimx. Request a buffer and frame status 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 */

typedef DEV_Callback SIO_Callback;

struct SIO_Attrs { /* stream attributes */
    Int    nbufs;      /* number of buffers */
    Int    segid;      /* buffer segment ID */
    size_t align;      /* buffer alignment */
    Bool   flush;      /* TRUE->don't block in DEV_idle*/
    Uns    model;      /* SIO_STANDARD, SIO_ISSUERECLAIM*/
    Uns    timeout;     /* passed to DEV_reclaim */
    SIO_Callback *callback;
                      /* initializes callback in DEV_Obj */
} SIO_Attrs;
```



```
SIO_Attrs SIO_ATTRS = {
    2,                /* nbufs */
    0,                /* segid */
    0,                /* align */
    FALSE,            /* flush */
    SIO_STANDARD,     /* model */
    SYS_FOREVER       /* timeout */
    NULL              /* callback */
};
```

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SIO Manager Properties and SIO Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

| Name            | Type      | Default           |
|-----------------|-----------|-------------------|
| OBJMEMSEG       | Reference | prog.get("IDRAM") |
| USEISSUERECLAIM | Bool      | false             |

**Instance Configuration Parameters**

| Name             | Type       | Default (Enum Options)              |
|------------------|------------|-------------------------------------|
| comment          | String     | "<add comments here>"               |
| deviceName       | Reference  | prog.get("dev-name")                |
| controlParameter | String     | ""                                  |
| mode             | EnumString | "input" ("output")                  |
| bufSize          | Int16      | 0x80                                |
| numBufs          | Int16      | 2                                   |
| bufSegId         | Reference  | prog.get("SIO.OBJMEMSEG")           |
| bufAlign         | EnumInt    | 1 (2, 4, 8, 16, 32, 64, ..., 32768) |
| flush            | Bool       | false                               |
| modelName        | EnumString | "Standard" ("Issue/Reclaim")        |
| allocStaticBuf   | Bool       | false                               |
| timeout          | Int16      | -1                                  |
| useCallBackFxn   | Bool       | false                               |
| callBackFxn      | Extern     | prog.extern("FXN_F_nop")            |
| arg0             | Arg        | 0                                   |
| arg1             | Arg        | 0                                   |

**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 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.

Both SWI and TSK threads can be used with the SIO module. However, SWI threads can be used only with the issue/reclaim model, and only then if the timeout parameter is 0. TSK threads can be used with either model.

**SIO Manager Properties**

The following global properties can be set for the SIO module in the SIO Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ☐ **Object Memory.** The memory segment that contains the SIO objects created with Tconf.

Tconf Name: `OBJMEMSEG`

Type: Reference

Example: `bios.SIO.OBJMEMSEG = prog.get("myMEM");`

- ☐ **Use Only Issue/Reclaim Model.** Enable this option if you want the SIO module to use only the issue/reclaim model. If this option is false (the default) you can also use the standard model.

Tconf Name: `USEISSUERECLAIM`

Type: Bool

Example: `bios.SIO.USEISSUERECLAIM = false;`

**SIO Object Properties**

To create an SIO object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySio = bios.SIO.create("mySio");
```

The following properties can be set for an SIO object in the SIO Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this SIO object.  
 Tconf Name: comment Type: String  
 Example: `mySio.comment = "my SIO";`
  
- ❑ **Device.** Select the device to which you want to bind this SIO object. User-defined devices are listed along with DGN and DPI devices.  
 Tconf Name: deviceName Type: Reference  
 Example: `mySio.deviceName = prog.get("UDEVO");`
  
- ❑ **Device Control String.** Type the device suffix to be passed to any devices stacked below the device connected to this stream.  
 Tconf Name: controlParameter Type: String  
 Example: `mySio.controlParameter =  
           "/split4/codec";`
  
- ❑ **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.  
 Tconf Name: mode Type: EnumString  
 Options: "input", "output"  
 Example: `mySio.mode = "input";`
  
- ❑ **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.  
 Tconf Name: bufSize Type: Int16  
 Example: `mySio.bufSize = 0x80;`
  
- ❑ **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.  
 Tconf Name: numBufs Type: Int16  
 Example: `mySio.numBufs = 2;`
  
- ❑ **Place buffers in memory segment.** Select the memory segment to contain the stream buffers if Model is Standard.  
 Tconf Name: bufSegId Type: Reference  
 Example: `mySio.bufSegId = prog.get("myMEM");`

- ☐ **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.

Tconf Name: bufAlign

Type: EnumInt

Options: 1, 2, 4, 8, 16, 32, 64, ..., 32768

Example: `mySio.bufAlign = 1;`

- ☐ **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.

Tconf Name: flush

Type: Bool

Example: `mySio.flush = false;`

- ☐ **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. Both SWI and TSK threads can be used with the SIO module. However, SWI threads can be used only with the issue/reclaim model, and only then if the timeout parameter is 0. TSK threads can be used with either model.

Tconf Name: modelName

Type: EnumString

Options: "Standard", "Issue/Reclaim"

Example: `mySio.modelName = "Standard";`

- ☐ **Allocate Static Buffer(s).** If this property is set to true, the configuration 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.

Tconf Name: allocStaticBuf

Type: Bool

Example: `mySio.allocStaticBuf = false;`

- ☐ **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.

Tconf Name: timeout

Type: Int16

Example: `mySio.timeout = -1;`

- ❑ **use callback function.** Check this box if you want to use this SIO object with a callback function. In most cases, the callback function is SWI\_andnHook or a similar function that posts a SWI. Checking this box allows the SIO object to be used with SWI threads.

Tconf Name: useCallBackFxn

Type: Bool

Example: `mySio.useCallBackFxn = false;`

- ❑ **callback function.** A function for the SIO object to call. In most cases, the callback function is SWI\_andnHook or a similar function that posts a SWI. This function gets called by the class driver (see the DIO Adapter) in the class driver's callback function. This callback function in the class driver usually gets called in the mini-driver code as a result of the HWI.

Tconf Name: callBackFxn

Type: Extern

Example: `mySio.callBackFxn =  
          prog.extern("SWI_andnHook");`

- ❑ **argument 0.** The first argument to pass to the callback function. If the callback function is SWI\_andnHook, this argument should be a SWI object handle.

Tconf Name: arg0

Type: Arg

Example: `mySio.arg0 = prog.get("mySwi");`

- ❑ **argument 1.** The second argument to pass to the callback function. If the callback function is SWI\_andnHook, this argument should be a value mask.

Tconf Name: arg1

Type: Arg

Example: `mySio.arg1 = 2;`

**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**              `size_t            size;`**Description**                `SIO_bufsize` returns the size of the buffers used by stream.

This API can be used only if the model is SIO\_STANDARD.

**See Also**                    `SIO_segid`

**SIO\_create***Open a stream***C Interface**

|                     |                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>stream = SIO_create(name, mode, bufsize, attrs);</code>                                                                                                                                                                               |
| <b>Parameters</b>   | <p>String        name;     /* name of device */</p> <p>Int           mode;     /* SIO_INPUT or SIO_OUTPUT */</p> <p>size_t        bufsize;  /* stream buffer size */</p> <p>SIO_Attrs     *attrs;    /* pointer to stream attributes */</p> |
| <b>Return Value</b> | <code>SIO_Handle stream;    /* stream object handle */</code>                                                                                                                                                                               |

**Description**

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).

Internally, SIO\_create calls Dxx\_open to open a device.

The mode parameter specifies whether the stream is to be used for input (SIO\_INPUT) or output (SIO\_OUTPUT).

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.

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.

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:

```
struct SIO_Attrs { /* stream attributes */
    Int     nbufs;      /* number of buffers */
    Int     segid;      /* buffer segment ID */
    size_t  align;      /* buffer alignment */
    Bool    flush; /* TRUE->don't block in DEV_idle */
    Uns     model; /* SIO_STANDARD, SIO_ISSUERECLAIM */
    Uns     timeout; /* passed to DEV_reclaim */
    SIO_Callback *callback;
                /* initialize callback in DEV_Obj */
} SIO_Attrs;
```

- ❑ **nbufs.** 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.
- ❑ **segid.** Specifies the memory segment for stream buffers. Use the memory segment names defined in the configuration. The default value is 0, meaning that buffers are to be allocated from the "Segment for DSP/BIOS objects" property in the MEM Manager Properties.
- ❑ **align.** Specifies the memory alignment for stream buffers. The default value is 0, meaning that no alignment is needed.
- ❑ **flush.** 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.
- ❑ **model.** 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.
- ❑ **timeout.** 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.



- ❑ **callback.** Specifies a pointer to channel-specific callback information. The SIO\_Callback structure is defined by the SIO module to match the DEV\_Callback structure. This structure contains the callback function and two function arguments. The callback function is typically SWI\_andnHook or a similar function that posts a SWI. Callbacks can only be used with the SIO\_ISSUERECLAIM model.

Existing DEV drivers do not use this callback function. While DEV drivers can be modified to use this callback, it is not recommended. Instead, the IOM device driver model is recommended for drivers that need the SIO callback feature. IOM drivers use the DIO module to interface with the SIO functions.

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 set by the "Segment for DSP/BIOS objects" property in the MEM Manager Properties.

## 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 Tconf.
- ❑ You can reduce the size of your application program by creating objects with Tconf 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 a 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 */<br>Uns cmd; /* command to device */<br>Arg arg; /* arbitrary argument */                                                                                                                                                                                                                          |
| Return Value                    | Int status; /* device status */                                                                                                                                                                                                                                                                                                          |
| Description                     | <p>SIO_ctrl causes a control operation to be issued to the device associated with stream. cmd and arg are passed directly to the device.</p> <p>SIO_ctrl returns SYS_OK if successful, and a non-zero device-dependent error value if unsuccessful.</p> <p>Internally, SIO_ctrl calls Dxx_ctrl to send control commands to a device.</p> |
| Constraints and Calling Context | <p>❑ SIO_ctrl cannot be called from an HWI.</p>                                                                                                                                                                                                                                                                                          |
| See Also                        | Dxx_ctrl                                                                                                                                                                                                                                                                                                                                 |

**SIO\_delete***Close a stream and free its buffers***C Interface**

**Syntax**                    `status = SIO_delete(stream);`

**Parameters**            `SIO_Handle stream;   /* stream object */`

**Return Value**          `Int                   status;   /* result of operation */`

**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 flush field in the SIO\_Attrs structure (passed in with SIO\_create). If flush is TRUE, SIO\_delete discards all pending data and returns 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 a 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 Tconf, SYS\_error is called.
- ❑ In SIO\_ISSUERECLAIM mode, all buffers issued to a stream must be reclaimed before SIO\_delete is called. Failing to reclaim such buffers causes a memory leak.

**See Also**

SIO\_create  
SIO\_flush  
SIO\_idle  
Dxx\_idle  
Dxx\_close

**SIO\_flush***Flush a stream***C Interface**

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| <b>Syntax</b>       | <code>status = SIO_flush(stream);</code>            |
| <b>Parameters</b>   | <code>SIO_Handle stream; /* stream handle */</code> |
| <b>Return Value</b> | <code>Int status; /* result of operation */</code>  |

**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.

If a callback was specified in the SIO\_Attrs structure used with SIO\_create, then SIO\_flush performs no processing and returns SYS\_OK.

**Constraints and  
Calling Context**

- ☐ SIO\_flush cannot be called from an HWI.
- ☐ If SIO\_flush is called from a SWI, no action is performed.

**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> <i>/* stream handle */</i><br><code>Ptr</code> <code>*bufp;</code> <i>/* pointer to a buffer */</i> |
| <b>Return Value</b> | <code>Int</code> <code>nmadus;</code> <i>/* number of MADUs read or error if negative */</i>                                       |

**Description**

SIO\_get exchanges an empty buffer with a non-empty buffer from stream. The bufp 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 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.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size\_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- ❑ **If the actual buffer size is less than/equal to the maximum positive Int value (31 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- ❑ **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. There is little room for this situation on 'C6000 since size\_t is the same as unsigned int, which is 32 bits. Since the sign in Int takes up one bit, the size\_t type contains just one more bit than an Int.

For other architectures, `size_t` is:

- ❑ 'C28x - unsigned long
- ❑ 'C54x/'C55x/'C6x - unsigned int

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 a SWI or HWI.
- ❑ This API is callable from the program's `main()` function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

### **See Also**

`Dxx_issue`  
`Dxx_reclaim`  
`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 */`

**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.

If a callback was specified in the SIO\_Attrs structure used with SIO\_create, then SIO\_idle performs no processing and returns SYS\_OK.

**Constraints and Calling Context**

- ☐ SIO\_idle cannot be called from an HWI.
- ☐ If SIO\_idle is called from a SWI, no action is performed.

**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 */
size_t      nmadus;  /* number of MADUs in the buffer */
Arg         arg;     /* user argument */
```

**Return Value**

```
Int         status;  /* result of operation */
```

**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.

---

**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 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 \*/  
`size_t nmadus;` /\* number of MADUs in the buffer \*/

**Return Value**

`Int nmadus;` /\* number of MADUs, negative if error \*/

**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).

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size\_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- ❑ **If the actual buffer size is less than/equal to the maximum positive Int value (31 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- ❑ **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. There is little room for this situation on 'C6000 since size\_t is the same as unsigned int, which is 32 bits. Since the sign in Int takes up one bit, the size\_t type contains just one more bit than an Int.

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 a SWI or HWI.
- ❑ This API is callable from the program's `main()` function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

### **See Also**

`Dxx_issue`  
`Dxx_reclaim`  
`SIO_get`

**SIO\_ready***Determine if device for stream is ready***C Interface****Syntax**                      `status = SIO_ready(stream);`**Parameters**                `SIO_Handle stream;`**Return Value**            `Int                status;    /* result of operation */`**Description**              `SIO_ready` returns TRUE if a stream is ready for input or output.

If you are using SIO objects with SWI threads, you may want to use `SIO_ready` to avoid calling `SIO_reclaim` when it may fail because no buffers are available.

`SIO_ready` is similar to `SIO_select`, except that it does not block. You can prevent `SIO_select` from blocking by setting the timeout to zero, however, `SIO_ready` is more efficient because `SIO_select` performs `SEM_pend` with a timeout of zero. `SIO_ready` simply polls the stream to see if the device is ready.

**See Also**                      `SIO_select`

**SIO\_reclaim***Request a buffer back from a stream***C Interface**

|                     |                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>nmadus = SIO_reclaim(stream, pbufp, parg);</code>                                                                                                                 |
| <b>Parameters</b>   | <code>SIO_Handle stream; /* stream handle */</code><br><code>Ptr *pbufp; /* pointer to the buffer */</code><br><code>Arg *parg; /* pointer to a user argument */</code> |
| <b>Return Value</b> | <code>Int nmadus; /* number of MADUs or error if negative */</code>                                                                                                     |

**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.

If you want to return a frame-specific status along with the buffer, use SIO\_reclaimx instead of SIO\_reclaim.

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

If a stream was created in SIO\_OUTPUT mode, then SIO\_reclaim returns an empty buffer, and nmadus is zero, since the buffer is empty. If a 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.

If SIO\_reclaim is called from a TSK thread, it blocks (in either mode) 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.

If SIO\_reclaim is called from a SWI thread, it returns an error if it is called when no buffer is available. SIO\_reclaim never blocks when called from a SWI.

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.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size\_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- ❑ **If the actual buffer size is less than/equal to the maximum positive Int value (31 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- ❑ **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. There is little room for this situation on 'C6000 since size\_t is the same as unsigned int, which is 32 bits. Since the sign in Int takes up one bit, the size\_t type contains just one more bit than an Int.

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.

---

**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.
- ❑ SIO\_reclaim returns an error if it is called from a SWI when no buffer is available. SIO\_reclaim does not block if called from a SWI.
- ❑ All frames issued to a stream must be reclaimed before closing the stream.
- ❑ SIO\_reclaim cannot be called from a HWI.
- ❑ This API is callable from the program's main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

**See Also**

Dxx\_reclaim  
SIO\_issue  
SIO\_create  
SIO\_reclaimx

**SIO\_reclaimx***Request a buffer back from a stream, including frame status***C Interface****Syntax**

```
nmadus = SIO_reclaimx(stream, *pbufp, *parg, *pfstatus);
```

**Parameters**

```
SIO_Handle stream; /* stream handle */  
Ptr        *pbufp; /* pointer to the buffer */  
Arg        *parg;  /* pointer to a user argument */  
Int        *pfstatus; /* pointer to frame status */
```

**Return Value**

```
Int          nmadus; /* number of MADUs or error if negative */
```

**Description**

SIO\_reclaimx is identical to SIO\_reclaim, except that it also returns a frame-specific status in the Int pointed to by the pfstatus parameter.

The device driver can use the frame-specific status to pass frame-specific status information to the application. This allows the device driver to fill in the status for each frame, and gives the application access to that status.

The returned frame status is valid only if SIO\_reclaimx() returns successfully. If the nmadus value returned is negative, the frame status should not be considered accurate.

**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\_reclaimx call is made.
- ❑ SIO\_reclaimx returns an error if it is called from a SWI when no buffer is available. SIO\_reclaimx does not block if called from a SWI.
- ❑ All frames issued to a stream must be reclaimed before closing the stream.
- ❑ SIO\_reclaimx cannot be called from a HWI.
- ❑ This API is callable from the program's main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

**See Also**

SIO\_reclaim



**SIO\_segid***Return the memory segment used by the stream***C Interface****Syntax**                      `segid = SIO_segid(stream);`**Parameters**                `SIO_Handle stream;`**Return Value**              `Int                segid;     /* memory segment ID */`**Description**                `SIO_segid` returns the identifier of the memory segment that stream uses for buffers.**See Also**                    `SIO_bufsize`

**SIO\_select***Select a ready device***C Interface****Syntax**

```
mask = SIO_select(streamtab, nstreams, timeout);
```

**Parameters**

```
SIO_Handle streamtab; /* stream table */
Int         nstreams; /* number of streams */
Uns         timeout;  /* return after this many system clock ticks */
```

**Return Value**

```
Uns         mask;     /* stream ready mask */
```

**Description**

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).

streamtab[] is an array of streams where nstreams < 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.

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.

SIO\_select results in a context switch if no streams are ready for I/O.

Internally, SIO\_select calls Dxx\_ready to determine if the device is ready for an I/O operation.

SIO\_ready is similar to SIO\_select, except that it does not block. You can prevent SIO\_select from blocking by setting the timeout to zero, however, SIO\_ready is more efficient in this situation because SIO\_select performs SEM\_pend with a timeout of zero. SIO\_ready simply polls the stream to see if the device is ready.

For the SIO\_STANDARD model in SIO\_INPUT mode only, if stream I/O has not been started (that is, if SIO\_get has not been called), SIO\_select calls Dxx\_issue for all empty frames to start the device.

**Constraints and  
Calling Context**

- ❑ streamtab must contain handles of type SIO\_Handle returned from prior calls to SIO\_create.
- ❑ streamtab[] is an array of streams; streamtab[i] corresponds to bit position i in mask.
- ❑ SIO\_select cannot be called from an HWI.
- ❑ SIO\_select can only be called from a SWI if the timeout value is zero.

**See Also**

Dxx\_ready  
SIO\_get  
SIO\_put  
SIO\_ready  
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 */
```

**Description**

SIO\_staticbuf returns buffers for static streams that were configured statically. 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.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size\_t, APIs that return a buffer size return a type of Int. This due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- ❑ **If the actual buffer size is less than/equal to the maximum positive Int value (31 bits).** Check the return value for negative values, which indicate errors. Positive values reflect the correct size.
- ❑ **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. There is little room for this situation on 'C6000 since size\_t is the same as unsigned int, which is 32 bits. Since the sign in Int takes up one bit, the size\_t type contains just one more bit than an Int.

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 Tconf.
- ❑ SIO\_staticbuf should only be called for static streams whose "Allocate Static Buffer(s)" property has been set to true.
- ❑ 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 HWI.

**See Also**

SIO\_get

2.28 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 */
}
```

Note:

STS objects should not be shared across threads. Therefore, STS\_add, STS\_delta, STS\_reset, and STS\_set are not reentrant.

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the STS Manager Properties and STS Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

Module Configuration Parameters

| Name      | Type      | Default           |
|-----------|-----------|-------------------|
| OBJMEMSEG | Reference | prog.get("IDRAM") |

Instance Configuration Parameters

| Name        | Type       | Default (Enum Options)                                                             |
|-------------|------------|------------------------------------------------------------------------------------|
| comment     | String     | "<add comments here>"                                                              |
| previousVal | Int32      | 0                                                                                  |
| unitType    | EnumString | "Not time based"<br>("High resolution time based",<br>"Low resolution time based") |
| operation   | EnumString | "Nothing" ("A * x", "A * x + B",<br>"(A * x + B) / C")                             |
| numA        | Int32      | 1                                                                                  |

| Name | Type  | Default (Enum Options) |
|------|-------|------------------------|
| numB | Int32 | 0                      |
| numC | Int32 | 1                      |

## 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 2-8.

*Table 2-8. 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 Tconf. 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

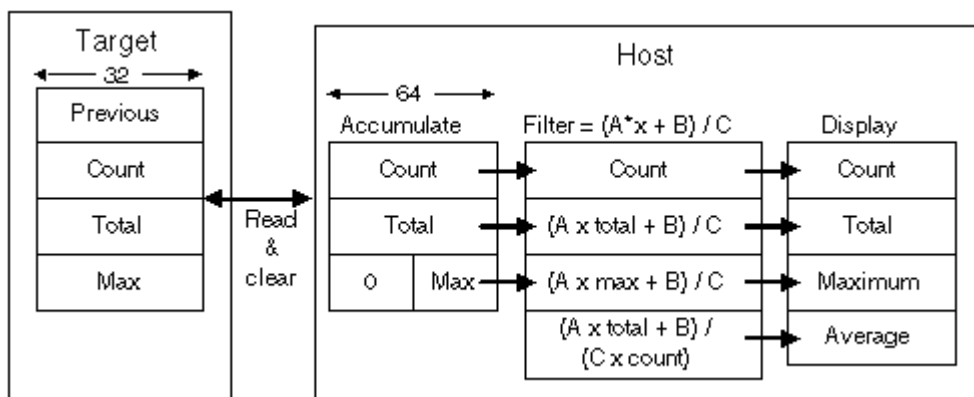


maximum value, the last 32-bit value passed to the object, the number of samples (up to  $2^{32} - 1$  samples), and the 32-bit sum of all samples.

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 Tconf, 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 2-8 shows the effects of the  $(A \times X + B) / C$  operation.

*Figure 2-8. Statistics Accumulation on the Host*



## STS Manager Properties

The following global property can be set for the STS module in the STS Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment that contains STS objects.

Tconf Name: OBJMEMSEG Type: Reference

Example: `bios.STS.OBJMEMSEG = prog.get("myMEM");`

## STS Object Properties

To create an STS object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySts = bios.STS.create("mySts");
```

The following properties can be set for an STS object in the STS Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this STS object.  
 Tconf Name: comment Type: String  
 Example: `mySts.comment = "my STS";`
- ❑ **prev.** The initial 32-bit history value to use in this object.  
 Tconf Name: previousVal Type: Int32  
 Example: `mySts.previousVal = 0;`
- ❑ **unit type.** The unit type property enables you to choose the type of time base units.
  - Not time based. If you select this unit type, the values are displayed in the Statistics View without applying any conversion.
  - High-resolution time based. If you select this type, the Statistics View, by default, presents results in units of instruction cycles.
  - Low-resolution time based. If you select this unit type, the default Statistics View presents results in timer interrupt units.
 Tconf Name: unitType Type: EnumString  
 Options: "Not time based", "High resolution time based", "Low resolution time based"  
 Example: `mySts.unitType = "Not time based";`
- ❑ **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:
  - $A \times X$
  - $A \times X + B$
  - $(A \times X + B) / C$
 Tconf Name: operation Type: EnumString  
 Options: "Nothing", "A \* x", "A \* x + B", "(A \* x + B) / C"  
 Example: `mySts.operation = "Nothing";`
- ❑ **A, B, C.** The integer parameters used by the expression specified by the Host Operation property above.  
 Tconf Name: numA Type: Int32  
 Tconf Name: numB Type: Int32  
 Tconf Name: numC Type: Int32  
 Example: `mySts.numA = 1;`  
           `mySts.numB = 0;`  
           `mySts.numC = 1;`

**STS\_add***Update statistics using the provided value***C Interface**

|                     |                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | STS_add(sts, value);                                                                                                              |
| <b>Parameters</b>   | STS_Handle sts;      /* statistics object handle */<br>LgInt                value;    /* new value to update statistics object */ |
| <b>Return Value</b> | Void                                                                                                                              |

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

|                    |                                                                                                      |
|--------------------|------------------------------------------------------------------------------------------------------|
| <b>Description</b> | 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.

|                 |                                                                |
|-----------------|----------------------------------------------------------------|
| <b>See Also</b> | STS_delta<br>STS_reset<br>STS_set<br>TRC_disable<br>TRC_enable |
|-----------------|----------------------------------------------------------------|

**STS\_delta***Update statistics using difference between provided value & 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**Reentrant** no**Description** Each STS object contains a previous value that can be initialized with Tconf 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 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 Tconf.

**Example**

```
STS_set(&sts, targetValue);  
    "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**

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| <b>Syntax</b>       | STS_reset(sts);                                     |
| <b>Parameters</b>   | STS_Handle sts;      /* statistics object handle */ |
| <b>Return Value</b> | Void                                                |

**Reentrant**                      no

**Description**                      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.

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.

**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**

|                     |                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | STS_set(sts, value);                                                                                                        |
| <b>Parameters</b>   | STS_Handle sts;       /* statistics object handle */<br>LgInt        value;     /* new value to update statistics object */ |
| <b>Return Value</b> | Void                                                                                                                        |
| <b>Reentrant</b>    | 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. 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
```

## 2.29 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 the mailbox value of the SWI when it started running.
- ❑ SWI\_getpri. Return a SWI's priority mask.
- ❑ SWI\_inc. Increment SWI's mailbox value and post the SWI.
- ❑ SWI\_isSWI. Check current thread calling context.
- ❑ SWI\_or. Or mask with value contained in SWI's mailbox and post the SWI.
- ❑ 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 a SWI's priority.
- ❑ SWI\_restorepri. Restore a SWI's priority.
- ❑ SWI\_self. Return address of currently executing SWI object.
- ❑ SWI\_setattrs. Set attributes of a software interrupt.

### Constants, Types, and Structures

```
typedef struct SWI_Obj SWI_Handle;

SWI_MINPRI  = 1;  /* Minimum execution priority */
SWI_MAXPRI  = 14  /* Maximum execution priority */
```



```
struct SWI_Attrs {      /* SWI attributes */
    SWI_Fxn  fxn;        /* address of SWI function */
    Arg      arg0;       /* first arg to function */
    Arg      arg1;       /* second arg to function */
    Int      priority;   /* Priority of SWI object */
    Uns      mailbox;    /* check for SWI posting */
};

SWI_Attrs SWI_ATTRS = { /* Default attribute values */
    (SWI_Fxn)FXN_F_nop,  /* SWI function */
    0,                  /* arg0 */
    0,                  /* arg1 */
    1,                  /* priority */
    0                   /* mailbox */
};
```

**Configuration Properties**

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SWI Manager Properties and SWI Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

| Name      | Type      | Default           |
|-----------|-----------|-------------------|
| OBJMEMSEG | Reference | prog.get("IDRAM") |

**Instance Configuration Parameters**

| Name     | Type    | Default (Enum Options)   |
|----------|---------|--------------------------|
| comment  | String  | "<add comments here>"    |
| fxn      | Extern  | prog.extern("FXN_F_nop") |
| priority | EnumInt | 1 (0 to 14)              |
| mailbox  | Int16   | 0                        |
| arg0     | Arg     | 0                        |
| arg1     | Arg     | 0                        |

**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 interrupt routines, 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: RTS Functions Callable from TSK Threads Only**

Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK\_pend and LCK\_post. As a result, RTS functions that call LCK\_pend or LCK\_post *must not be called in the context of a SWI or HWI thread*. For a list of RTS functions that should not be called from a SWI or an HWI function, see “LCK\_pend” on page 2-210.

---

The C++ new operator calls malloc, which in turn calls LCK\_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

The processor registers that are saved before SWI functions are called include a0-a9 and b0-b9. These registers are the parent-preserved registers mentioned in the *TMS320C6000 Optimizing Compiler User's Guide*. The child-preserved registers, a10-a15 and b10-b15, are not saved.

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 a SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a SWI 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 SWI's priority to determine whether to preempt the thread currently running. Note that if a SWI is posted several times before it begins running, (because HWIs and higher priority interrupts are running,) when the SWI does eventually run, it will run only one time.

Software interrupts can be posted for execution with a call to SWI\_post or a number of other SWI functions. Each SWI object has a 32-bit mailbox which is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI's function. SWI\_andn and SWI\_dec post the SWI 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 SWIs and enable them all for execution at the same time. The SWI priorities then determine which SWI runs first.

All SWIs run to completion; you cannot suspend a SWI while it waits for something (for example, a device) to be ready. So, you can use the mailbox to tell the SWI when all the devices and other conditions it relies on are ready. Within a SWI processing function, a call to SWI\_getmbx returns the value of the mailbox when the SWI started running. Note that the mailbox is automatically reset to its original value when a SWI runs; however, SWI\_getmbx will return the saved mailbox value from when the SWI started execution.

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 (TSK) scheduler.

A SWI preempts any currently running SWI with a lower priority. If two SWIs with the same priority level have been posted, the SWI that was posted first runs first. HWIs in turn preempt any currently running SWI, allowing the target to respond quickly to hardware peripherals.

Interrupt threads (including HWIs and SWIs) 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 SWI that preempts a lower-priority SWI. After the higher-priority SWI finishes running, the registers are restored and the lower-priority SWI can run if no other higher-priority SWI has 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 SWIs and scheduling issues for the Software Interrupt manager.

## SWI Manager Properties

The following global property can be set for the SWI module in the SWI Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **Object Memory.** The memory segment that contains the SWI objects.

Tconf Name: OBJMEMSEG

Type: Reference

Example: `bios.SWI.OBJMEMSEG = prog.get("myMEM");`

## SWI Object Properties

To create a SWI object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySwi = bios.SWI.create("mySwi");
```

If you cannot create a new SWI object (an error occurs or the Insert SWI item is inactive in the DSP/BIOS Configuration Tool), try increasing the Stack Size property in the MEM Manager Properties before adding a SWI object or a SWI priority level.

The following properties can be set for a SWI object in the SWI Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **comment.** Type a comment to identify this SWI object.

Tconf Name: comment

Type: String

Example: `mySwi.comment = "my SWI";`

- ❑ **function.** The function to execute. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. (The DSP/BIOS Configuration Tool generates assembly code, which must use leading underscores when referencing C functions or labels.) If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

Tconf Name: fxn

Type: Extern

Example: `mySwi.fxn = prog.extern("swiFxn");`

- ❑ **priority.** This property shows the numeric priority level for this SWI object. SWIs 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 the DSP/BIOS Configuration Tool, you change the relative priority levels of SWI objects by dragging the objects in the ordered collection view.

Tconf Name: priority

Type: EnumInt

Options: 0 to 14

Example: `mySwi.priority = 1;`

- ❑ **mailbox.** The initial value of the 32-bit word used to determine if this SWI should be posted.

Tconf Name: mailbox

Type: Int16

Example: `mySwi.mailbox = 7;`

- ❑ **arg0, arg1.** Two arbitrary pointer type (Arg) arguments to the above configured user function.

Tconf Name: arg0

Type: Arg

Tconf Name: arg1

Type: Arg

Example: `mySwi.arg0 = 0;`

**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*/<br>Uns           mask     /* inverse value to be ANDed */                                                                                                                                 |
| <b>Return Value</b> | Void                                                                                                                                                                                                                                   |
| <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 SWI. The bitwise logical operation performed is: |

```
mailbox = mailbox AND (NOT MASK)
```

For example, if multiple conditions that all be met before a SWI 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 SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.

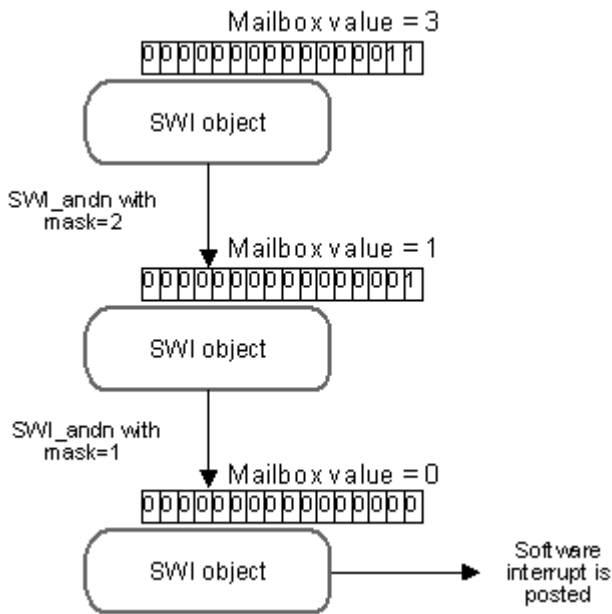
---

**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 function is invoked outside the context of an HWI, interrupts must be enabled.
- ❑ When called within an HWI, 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****Syntax** SWI\_andnHook(swi, mask);**Parameters**  
Arg swi; /\* SWI object handle\*/  
Arg mask /\* value to be ANDed \*/**Return Value** Void**Reentrant** yes**Description**  
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 SWI. 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 SWI 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 SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.

**Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- ❑ When called within an HWI, 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_andnHook(&copySWI, mask);  
}
```

**See Also**SWI\_andn  
SWI\_orHook



**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>  |

**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. The SWI object's attributes are specified through a structure of type SWI\_attr defined as follows:

```
struct SWI_Attrs {
    SWI_Fxn  fxn;
    Arg      arg0;
    Arg      arg1;
    Int      priority;
    Uns      mailbox;
};
```

If attrs is NULL, the new SWI object is assigned the following default attributes.

```
SWI_Attrs SWI_ATTRS = { /* Default attribute values */
    (SWI_Fxn)FXN_F_nop, /* SWI function */
    0,                  /* arg0 */
    0,                  /* arg1 */
    1,                  /* priority */
    0                   /* 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 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 2–240.

### **Constraints and Calling Context**

- ❑ SWI\_create cannot be called from a 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                                        |

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

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Description</b> | <p>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 SWI. You can increment a mailbox value by using SWI_inc, which always posts the SWI.</p> <p>For example, you would use SWI_dec if you wanted to post a SWI after a number of occurrences of an event.</p> <p>You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.</p> <p>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.</p> |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- ❑ When called within an HWI, 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\_inc

**SWI\_delete***Delete a software interrupt***C Interface****Syntax**

SWI\_delete(swi);

**Parameters**

SWI\_Handle swi;       /\* SWI object handle \*/

**Return Value**

Void

**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 a 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 Tconf, 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**Reentrant** yes

**Description** SWI\_disable and SWI\_enable control software interrupt processing. SWI\_disable disables all other SWI functions from running until SWI\_enable is called. Hardware interrupts can still run.

SWI\_disable and SWI\_enable let you 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 SWIs.

```
SWI_disable();
    ~critical section~
SWI_enable();
```

You can also use SWI\_disable and SWI\_enable to post several SWIs and have them performed in priority order. See the following example.

SWI\_disable calls can be nested. The number of nesting levels is stored internally. SWI 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 HWIs that schedule SWIs automatically disable and reenable SWI handling. You should not call SWI\_disable or SWI\_enable within a HWI.
- ❑ SWI\_disable cannot be called from the program's main() function.
- ❑ Do not call SWI\_enable when SWIs are already enabled. If you do, a subsequent call to SWI\_disable does not disable SWI processing.

**Example**

```
/* ===== postEm ===== */
Void postEm
{
    SWI_disable();
    SWI_post(&encoderSwi);
    SWI_andn(&copySwi, mask);
    SWI_dec(&strikeoutSwi);
    SWI_enable();
}
```

**See Also**

HWI\_disable  
SWI\_enable

SWI\_enable

Enable software interrupts

C Interface

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax                          | SWI_enable();                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Parameters                      | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Return Value                    | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Reentrant                       | yes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description                     | <p>SWI_disable and SWI_enable control software interrupt processing. SWI_disable disables all other SWI 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. SWI handling is not be reenabled until SWI_enable has been called as many times as SWI_disable.</p> <p>SWI_enable results in a context switch if a higher-priority SWI is ready to run.</p> |
| Constraints and Calling Context | <ul style="list-style-type: none"><li>❑ The calls to HWI_enter and HWI_exit are required in any HWI that schedules SWIs. They automatically disable and reenable SWI handling. You should not call SWI_disable or SWI_enable within a HWI.</li><li>❑ SWI_enable cannot be called from the program's main() function.</li><li>❑ Do not call SWI_enable when SWIs are already enabled. If you do so, the subsequent call to SWI_disable will not disable SWI processing.</li></ul>                                              |
| See Also                        | <p>HWI_disable<br/>HWI_enable<br/>SWI_disable</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

**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 */<br>SWI_Attrs *attrs;    /* pointer to swi attributes */ |
| <b>Return Value</b> | Void                                                                                                  |

**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 a 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 */`

**Reentrant**                   `yes`

**Description**

SWI\_getmbox returns the value that SWI's mailbox had when the SWI started running. DSP/BIOS saves the mailbox value internally so that SWI\_getmbox can access it at any point within a SWI object's function. DSP/BIOS then automatically resets the mailbox to its initial value (defined with Tconf) so that other threads can continue to use the SWI's mailbox.

SWI\_getmbox should only be called within a function run by a SWI object.

When called from with the context of a SWI, the value returned by SWI\_getmbox is zero if the SWI was posted by a call to SWI\_andn, SWI\_andnHook, or SWI\_dec. Therefore, SWI\_getmbox provides relevant information only if the SWI was posted by a call to SWI\_inc, SWI\_or, SWI\_orHook, or SWI\_post.

**Constraints and Calling Context**

- ❑ SWI\_getmbox cannot be called from the context of an HWI or TSK.
- ❑ SWI\_getmbox cannot be called from a 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 SWI, 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

|              |                                                                                                                                                                            |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | key = SWI_getpri(swi);                                                                                                                                                     |
| Parameters   | SWI_Handle swi;      /* SWI object handle*/                                                                                                                                |
| Return Value | Uns            key      /* Priority mask of swi */                                                                                                                         |
| Reentrant    | yes                                                                                                                                                                        |
| Description  | SWI_getpri returns the priority mask of the SWI passed in as the argument.                                                                                                 |
| Example      | <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> |
| See Also     | SWI_raisepri<br>SWI_restorepri                                                                                                                                             |

**SWI\_inc***Increment SWI's mailbox value and post the SWI***C Interface**

|                     |                                             |
|---------------------|---------------------------------------------|
| <b>Syntax</b>       | SWI_inc(swi);                               |
| <b>Parameters</b>   | SWI_Handle swi;      /* SWI object handle*/ |
| <b>Return Value</b> | Void                                        |

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

**Description**

SWI\_inc increments the value in SWI's mailbox by 1 and posts the SWI regardless of the resulting mailbox value. You can decrement a mailbox value using SWI\_dec, which only posts the SWI if the mailbox value is 0.

If a SWI is posted several times before it has a chance to begin executing, because HWIs and higher priority SWIs are running, the SWI only runs one time. If this situation occurs, you can use SWI\_inc to post the SWI. Within the SWI's function, you could then use SWI\_getmbox to find out how many times this SWI has been posted since the last time it was executed.

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI 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 Hwi, interrupts must be enabled.
- ❑ When called within an Hwi, 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\_dec  
SWI\_getmbox

**SWI\_isSWI***Check to see if called in the context of a SWI***C Interface****Syntax**

```
result = SWI_isSWI(Void);
```

**Parameters**

Void

**Return Value**

Bool            result;     /\* TRUE if in SWI context, FALSE otherwise \*/

**Reentrant**

yes

**Description**

This macro returns TRUE when it is called within the context of a SWI or PRD function. This applies no matter whether the SWI was posted by an HWI, TSK, or IDL thread. This macro returns FALSE in all other contexts.

In previous versions of DSP/BIOS, calling SWI\_isSWI() from a task switch hook resulted in TRUE. This is no longer the case; task switch hooks are identified as part of the TSK context.

**See Also**

HWI\_isHWI

TSK\_isTSK

**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*/<br>Uns            mask;    /* value to be ORed */ |
| <b>Return Value</b> | Void                                                                                          |
| <b>Reentrant</b>    | no                                                                                            |

**Description**

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 SWI regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes. To get the mailbox value, use SWI\_getmbox.

For example, you might use SWI\_or to post a SWI if any of three events should cause a SWI to be executed, but you want the SWI'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 HWI, interrupts must be enabled.
- ☐ When called within an HWI, 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\_orHook

SWI\_orHook

OR mask with the value contained in SWI's mailbox field

C Interface

|              |                        |       |                        |
|--------------|------------------------|-------|------------------------|
| Syntax       | SWI_orHook(swi, mask); |       |                        |
| Parameters   | Arg                    | swi;  | /* SWI object handle*/ |
|              | Arg                    | mask; | /* value to be ORed */ |
| Return Value | Void                   |       |                        |
| Reentrant    | no                     |       |                        |

**Description** 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 SWI regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes. To get the mailbox value, use SWI\_getmbox.

For example, you might use SWI\_orHook to post a SWI if any of three events should cause a SWI to be executed, but you want the SWI'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 HWI, interrupts must be enabled.
- ❑ When called within an HWI, 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\_andnHook  
SWI\_or

**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                                        |

**Reentrant**                      yes

**Description**                      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.

To have a PRD object post a SWI object's function, you can set \_SWI\_post as the function property of a PRD object and the name of the SWI 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 HWI, interrupts must be enabled.
- ❑ When called within an HWI, 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\_getmbx  
SWI\_inc  
SWI\_or  
SWI\_self

**SWI\_raisepri***Raise a SWI's priority***C Interface**

|                     |                                                                 |
|---------------------|-----------------------------------------------------------------|
| <b>Syntax</b>       | <code>key = SWI_raisepri(mask);</code>                          |
| <b>Parameters</b>   | Uns            mask;     /* mask of desired priority level */   |
| <b>Return Value</b> | Uns            key;       /* key for use with SWI_restorepri */ |

**Reentrant**            yes

**Description**            SWI\_raisepri is used to raise the priority of the currently running SWI to the priority mask passed in as the argument. SWI\_raisepri can be used in conjunction with SWI\_restorepri to provide a mutual exclusion mechanism without disabling SWIs.

SWI\_raisepri should be called before a 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). Priority zero (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**

**Syntax**                      SWI\_restorepri(key);

**Parameters**                Uns                key;                /\* key to restore original priority level \*/

**Return Value**               Void

**Reentrant**                    yes

**Description**                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 SWIs.

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****Syntax**

```
curswi = SWI_self();
```

**Parameters**

Void

**Return Value**

SWI\_Handle swi;       /\* handle for current swi object \*/

**Reentrant**

yes

**Description**

SWI\_self returns the address of the currently executing SWI.

**Constraints and  
Calling Context**

- ❑ SWI\_self cannot be called from an HWI or TSK level.
- ❑ SWI\_self cannot be called from the program's main() function.

**Example**

You can use SWI\_self if you want a SWI to repost itself:

```
SWI_post(SWI_self());
```

**See Also**

```
SWI_andn  
SWI_getmbox  
SWI_post
```

**SWI\_setattrs***Set attributes of a software interrupt***C Interface**

|                     |                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | SWI_setattrs(swi, attrs);                                                                              |
| <b>Parameters</b>   | SWI_Handle swi;       /* handle of the swi */<br>SWI_Attrs *attrs;     /* pointer to swi attributes */ |
| <b>Return Value</b> | Void                                                                                                   |

**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 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

## 2.30 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 alloc error */
#define SYS_EFREE      2  /* memory free error */
#define SYS_ENODEV     3  /* dev driver not found */
#define SYS_EBUSY      4  /* device driver busy */
#define SYS_EINVAL     5  /* invalid parameter */
#define SYS_EBADIO     6  /* I/O failure */
#define SYS_EMODE      7  /* bad mode for 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 /* deleted obj */
#define SYS_EBADOBJ    12 /* invalid object */
#define SYS_ENOTIMPL   13 /* action not implemented */
#define SYS_ENOTFOUND  14 /* resource not found */

#define SYS_EUSER      256 /* user errors start here */

#define SYS_NUMHANDLERS 8 /* # of atexit handlers */

extern String SYS_errors[]; /* error string array */
```

### Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SYS Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

## Module Configuration Parameters

| Name      | Type      | Default                    |
|-----------|-----------|----------------------------|
| TRACESIZE | Numeric   | 512                        |
| TRACESEG  | Reference | prog.get("IDRAM")          |
| ABORTFXN  | Extern    | prog.extern("UTL_doAbort") |
| ERRORFXN  | Extern    | prog.extern("UTL_doError") |
| EXITFXN   | Extern    | prog.extern("UTL_halt")    |
| PUTCFXN   | Extern    | prog.extern("UTL_doPutc")  |

### 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 in the SYS Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script.

- ❑ **Trace Buffer Size.** The size of the buffer that contains system trace information. This system trace buffer can be viewed only by looking for the `SYS_PUTCBEG` symbol in the Code Composer Studio memory view. For example, by default the `Putc` function writes to the trace buffer.

Tconf Name: TRACESIZE

Type: Numeric

Example: `bios.SYS.TRACESIZE = 512;`

- ❑ **Trace Buffer Memory.** The memory segment that contains system trace information.

Tconf Name: TRACESEG

Type: Reference

Example: `bios.SYS.TRACESEG = prog.get ("myMEM") ;`

- ❑ **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 you are using `Tconf`, do not add an underscore before the function name; `Tconf` adds the underscore needed to call a C function from assembly internally. The prototype for this function should be:

```
Void myAbort(String fmt, va_list ap);
```

Tconf Name: ABORTFXN Type: Extern

```
Example:  bios.SYS.ABORTFXN =
           prog.extern("myAbort");
```

- ❑ **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 and returns. The prototype for this function should be:

```
Void myError(String s, Int errno, va_list ap);
```

Tconf Name: ERRORFXN Type: Extern

```
Example:  bios.SYS.ERRORFXN =
           prog.extern("myError");
```

- ❑ **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. The prototype for this function should be:

```
Void myExit(Int status);
```

Tconf Name: EXITFXN Type: Extern

```
Example:  bios.SYS.EXITFXN =
           prog.extern("myExit");
```

- ❑ **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 system trace buffer. This system trace buffer can be viewed only by looking for the `SYS_PUTCBEG` symbol in the Code Composer Studio memory view. The prototype for this function should be:

```
Void myPutc(Char c);
```

Tconf Name: PUTCFXN Type: Extern

```
Example:  bios.SYS.PUTCFXN =
           prog.extern("myPutc");
```

## SYS Object Properties

The SYS module does not support the creation of individual SYS objects.

**SYS\_abort***Abort program execution***C Interface****Syntax**`SYS_abort(format, [arg,] ...);`**Parameters**

|        |         |                                   |
|--------|---------|-----------------------------------|
| String | format; | /* format specification string */ |
| Arg    | arg;    | /* optional argument */           |

**Return Value**

Void

**Description**

SYS\_abort aborts program execution by calling the function bound to the configuration parameter Abort function, where vargs is of type va\_list (a void pointer which can be interpreted as an argument list) and represents the sequence of arg parameters originally passed to SYS\_abort.

```
(* (Abort_function)) (format, vargs)
```

The function bound to Abort function can elect to pass the format and vargs 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



**SYS\_atexit***Stack an exit handler***C Interface**

|                     |                                                                            |
|---------------------|----------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>success = SYS_atexit(handler);</code>                                |
| <b>Parameters</b>   | <code>Fxn            handler    /* exit handler function */</code>         |
| <b>Return Value</b> | <code>Bool            success    /* handler successfully stacked */</code> |

**Description**

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:

```
(*handler) (status)
```

SYS\_atexit returns TRUE if handler has been successfully stacked; FALSE if the internal stack is full.

The handlers on the stack are called only if either of the following happens:

- ☐ SYS\_exit is called.
- ☐ 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.)
- ☐ handler cannot be NULL.

**Constraints and  
Calling Context**

**SYS\_error**

*Flag error condition*

**C Interface**

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                         |    |                    |     |        |                  |     |      |                         |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|----|--------------------|-----|--------|------------------|-----|------|-------------------------|
| <b>Syntax</b>                          | SYS_error(s, errno, [arg], ...);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                         |    |                    |     |        |                  |     |      |                         |
| <b>Parameters</b>                      | <table><tr><td>String</td><td>s;</td><td>/* error string */</td></tr><tr><td>Int</td><td>errno;</td><td>/* error code */</td></tr><tr><td>Arg</td><td>arg;</td><td>/* optional argument */</td></tr></table>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | String                  | s; | /* error string */ | Int | errno; | /* error code */ | Arg | arg; | /* optional argument */ |
| String                                 | s;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | /* error string */      |    |                    |     |        |                  |     |      |                         |
| Int                                    | errno;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | /* error code */        |    |                    |     |        |                  |     |      |                         |
| Arg                                    | arg;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | /* optional argument */ |    |                    |     |        |                  |     |      |                         |
| <b>Return Value</b>                    | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                         |    |                    |     |        |                  |     |      |                         |
| <b>Description</b>                     | <p>SYS_error is used to flag DSP/BIOS error conditions. Application programs should call SYS_error to handle program errors. Internal functions also call SYS_error.</p> <p>SYS_error calls a function to handle errors. The default error function for the SYS manager is _UTL_doError, which logs an error message and returns. The default function can be replaced with your own error function by setting the SYS.ERRORFXN configuration property.</p> <p>The default error function or an alternate configured error function is called as follows, where vargs is of type va_list (a void pointer which can be interpreted as an argument list) and represents the sequence of arg parameters originally passed to SYS_error.</p> <pre>(*(Error_function))(s, errno, vargs)</pre> |                         |    |                    |     |        |                  |     |      |                         |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ 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.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                         |    |                    |     |        |                  |     |      |                         |

**SYS\_exit***Terminate program execution***C Interface**

|                     |                                                                      |
|---------------------|----------------------------------------------------------------------|
| <b>Syntax</b>       | <code>SYS_exit(status);</code>                                       |
| <b>Parameters</b>   | <code>Int            status;    /* termination status code */</code> |
| <b>Return Value</b> | <code>Void</code>                                                    |

**Description**

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.

```
(*handlerN) (status)
...
(*handler2) (status)
(*handler1) (status)

(* (Exit_function) ) (status)
```

The default Exit function for the SYS manager is UTL\_halt, which performs an infinite loop with all processor interrupts disabled.

**Constraints and Calling Context**

- ❑ If the function bound to Exit function or any of the handler functions is not reentrant, SYS\_exit must be called atomically.

**See Also**

SYS\_abort  
SYS\_atexit

SYS\_printf

Output formatted data

C Interface

|              |                                                                                                                       |
|--------------|-----------------------------------------------------------------------------------------------------------------------|
| Syntax       | SYS_printf(format, [arg,] ...);                                                                                       |
| Parameters   | String           format;     /* format specification string */<br>Arg             arg;        /* optional argument */ |
| Return Value | Void                                                                                                                  |
| 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 the 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 2-9.

Table 2-9. 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         | pointer                     |

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the 'C67x and 283xx).

Between the % and the conversion character, the following symbols or specifiers contained in 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 40-bit long integer. If the argument is a 32-bit long integer (`LgInt` or `LgUns`), the `l` modifier should not be used.

`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 via the `Putc` function configured in the SYS Manager Properties. The default `Putc` function is `_UTL_doPutc`, which writes a character to the system trace buffer. The size and memory segment for the system trace buffer can also be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the `SYS_PUTCBEG` symbol in the Code Composer Studio memory view.

## Constraints and Calling Context

- ❑ On a DSP with floating-point support, `SYS_printf` prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as `LONG_MAX` in the `<limits.h>` ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- ❑ On a DSP with floating-point support, `SYS_printf` only prints four digits after the decimal point for floating point numbers. Since `SYS_printf` does not support `%e`, floating point numbers have to be scaled approximately before being passed to `SYS_printf`.
- ❑ The function bound to Exit function or any of the handler functions are not reentrant; `SYS_exit` must be called atomically.

## See Also

`SYS_sprintf`  
`SYS_vprintf`  
`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                                                                                               |         |                                   |
| 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 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 2-10.

Table 2-10. 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         | pointer                     |

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the 'C67x and 283xx).

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 40-bit long integer. If the argument is a 32-bit long integer (LgInt or LgUns), the l modifier should not be used.

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

- ❑ On a DSP with floating-point support, SYS\_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG\_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- ❑ On a DSP with floating-point support, SYS\_printf only prints four digits after the decimal point for floating point numbers. Since SYS\_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS\_printf.
- ❑ 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

|              |                                                                                                                                   |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | SYS_vprintf(format, vars);                                                                                                        |
| Parameters   | String           format;   /* format specification string */<br>va_list          vars;     /* variable argument list reference */ |
| Return Value | Void                                                                                                                              |
| 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 2-11.

Table 2-11. 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         | pointer                     |

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the 'C67x and 283xx).



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 40-bit long integer. If the argument is a 32-bit long integer (LgInt or LgUns), the l modifier should not be used.

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 via the Putc function configured in the SYS Manager Properties. The default Putc function is \_UTL\_doPutc, which writes a character to the system trace buffer. The size and memory segment for the system trace buffer can also be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the SYS\_PUTCBEG symbol in the Code Composer Studio memory view.

## Constraints and Calling Context

- ❑ On a DSP with floating-point support, SYS\_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG\_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- ❑ On a DSP with floating-point support, SYS\_printf only prints four digits after the decimal point for floating point numbers. Since SYS\_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS\_printf.
- ❑ 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

|              |                                                                                                     |         |                                        |
|--------------|-----------------------------------------------------------------------------------------------------|---------|----------------------------------------|
| Syntax       | SYS_vsprintf(buffer, format, vars);                                                                 |         |                                        |
| Parameters   | String                                                                                              | buffer; | /* output buffer */                    |
|              | String                                                                                              | format; | /* format specification string */      |
|              | va_list                                                                                             | vars;   | /* variable argument list reference */ |
| Return Value | Void                                                                                                |         |                                        |
| 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 2-12.

Table 2-12. 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         | pointer                     |

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the 'C67x and 283xx).

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 40-bit long integer. If the argument is a 32-bit long integer (LgInt or LgUns), the l modifier should not be used.

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

- ❑ On a DSP with floating-point support, SYS\_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG\_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- ❑ On a DSP with floating-point support, SYS\_printf only prints four digits after the decimal point for floating point numbers. Since SYS\_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS\_printf.
- ❑ 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**

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>SYS_putchar(c);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Parameters</b>                      | <code>Char            c;            /* next output character */</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Return Value</b>                    | <code>Void</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>                     | <p>SYS_putchar outputs the character <code>c</code> by calling the system-dependent function bound to the configuration parameter <code>Putc</code> function.</p> <pre>((Putc function))(c)</pre> <p>For systems with limited I/O capabilities, the function bound to <code>Putc</code> function might simply place <code>c</code> into a global buffer that can be examined after program termination.</p> <p>The default <code>Putc</code> function for the SYS manager is <code>_UTL_doPutc</code>, which writes a character to the system trace buffer. The size and memory segment for the system trace buffer can be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the <code>SYS_PUTCBEG</code> symbol in the Code Composer Studio memory view.</p> <p>SYS_putchar is also used internally by <code>SYS_printf</code> and <code>SYS_vprintf</code> when generating their output.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ If the function bound to <code>Putc</code> function is not reentrant, SYS_putchar must be called atomically.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>See Also</b>                        | <code>SYS_printf</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## 2.31 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 2-13 lists events and statistics that can be traced. The constants defined in trc.h, trc.h62, and trc.h64 are shown in the left column.

*Table 2-13. 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 SWI 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<br>and<br>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\_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                                                   |

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

**Description** TRC\_disable disables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module topic 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

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | TRC_enable(mask);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Parameters   | Uns            mask;    /* trace type constant mask */                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Return Value | Void                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Reentrant    | no                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description  | <p>TRC_enable enables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module topic for a list of constants to use in the mask.)</p> <p>The following C code would enable tracing of statistics for software interrupts and periodic functions:</p> <pre>TRC_enable (TRC_STSSWI   TRC_STSPRD) ;</pre> <p>Internally, DSP/BIOS uses a bitwise OR operation to enable multiple trace types.</p> <p>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.</p> |
| See Also     | <p>TRC_disable<br/>TRC_query<br/>LOG_printf<br/>LOG_event<br/>STS_add<br/>STS_delta</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |



**TRC\_query***Query trace class(es)***C Interface**

|                     |                                        |        |                                                        |
|---------------------|----------------------------------------|--------|--------------------------------------------------------|
| <b>Syntax</b>       | <code>result = TRC_query(mask);</code> |        |                                                        |
| <b>Parameters</b>   | Uns                                    | mask;  | <i>/* trace type constant mask */</i>                  |
| <b>Return Value</b> | Int                                    | result | <i>/* indicates whether all trace types enabled */</i> |
| <b>Reentrant</b>    | 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 topic for a list of constants to use in the mask.)

Trace types are specified with a 32-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, this 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

## 2.32 TSK Module

The TSK module is the task manager.

### 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\_isTSK. Check current thread calling context
- ❑ 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(Void);  
Void TSK_readyFxn(TSK_Handle newtask);  
Void TSK_switchFxn(TSK_Handle oldtask,  
                  TSK_Handle newtask);
```

## Constants, Types, and Structures

```
typedef struct TSK_OBJ *TSK_Handle; /* task object handle*/

struct TSK_Attrs { /* task attributes */
    Int    priority; /* execution priority */
    Ptr    stack;    /* pre-allocated stack */
    size_t stacksize; /* stack size in MADUs */
    Int    stackseg; /* mem seg for stack allocation */
    Ptr    environ;  /* global environment data struct */
    String name;     /* printable name */
    Bool   exitflag; /* program termination requires */
                /* this task to terminate */
    Bool   initstackflag; /* initialize task stack? */
};

Int TSK_pid; /* MP processor ID */

Int TSK_MAXARGS = 8; /* max 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 TSK_STACKSTAMP = 0xBEBEBEBE
TSK_Attrs TSK_ATTRS = { /* default attribute values */
    TSK->PRIORITY, /* priority */
    NULL,          /* stack */
    TSK->STACKSIZE, /* stacksize */
    TSK->STACKSEG, /* stackseg */
    NULL,          /* environ */
    "",            /* name */
    TRUE,          /* exitflag */
    TRUE,          /* initstackflag */
};

enum TSK_Mode { /* task execution modes */
    TSK_RUNNING, /* task currently executing */
    TSK_READY,   /* task scheduled for execution */
    TSK_BLOCKED, /* task suspended from execution */
    TSK_TERMINATED, /* task 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 */
    size_t    used;  /* task stack used */
};
```

## Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the TSK Manager Properties and TSK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-4.

**Module Configuration Parameters**

| <b>Name</b>   | <b>Type</b> | <b>Default (Enum Options)</b> |
|---------------|-------------|-------------------------------|
| ENABLETSK     | Bool        | true                          |
| OBJMEMSEG     | Reference   | prog.get("IDRAM")             |
| STACKSIZE     | Int16       | 1024                          |
| STACKSEG      | Reference   | prog.get("IDRAM")             |
| PRIORITY      | EnumInt     | 1 (1 to 15)                   |
| DRIVETSKTICK  | EnumString  | "PRD" ("User")                |
| CREATEFXN     | Extern      | prog.extern("FXN_F_nop")      |
| DELETEFXN     | Extern      | prog.extern("FXN_F_nop")      |
| EXITFXN       | Extern      | prog.extern("FXN_F_nop")      |
| CALLSWITCHFXN | Bool        | false                         |
| SWITCHFXN     | Extern      | prog.extern("FXN_F_nop")      |
| CALLREADYFXN  | Bool        | false                         |
| READYFXN      | Extern      | prog.extern("FXN_F_nop")      |

**Instance Configuration Parameters**

| <b>Name</b>       | <b>Type</b> | <b>Default (Enum Options)</b> |
|-------------------|-------------|-------------------------------|
| comment           | String      | "<add comments here>"         |
| autoAllocateStack | Bool        | true                          |
| manualStack       | Extern      | prog.extern("null","asm")     |
| stackSize         | Int16       | 1024                          |
| stackMemSeg       | Reference   | prog.get("IDRAM")             |
| priority          | EnumInt     | 0 (-1, 0, 1 to 15)            |
| fxn               | Extern      | prog.extern("FXN_F_nop")      |
| arg0              | Arg         | 0                             |
| arg7              | Arg         | 0                             |
| envPointer        | Arg         | 0x00000000                    |
| exitFlag          | Bool        | true                          |
| allocateTaskName  | Bool        | false                         |
| order             | Int16       | 0                             |

## 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 a 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 `TSK_STACKSTAMP` value is used to initialize the run-time stack. When creating a task dynamically, you need to initialize the stack with `TSK_STACKSTAMP` only if the stack is allocated manually and `TSK_checkstacks` or `TSK_stat` is to be called. 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 the SEM Module and the SIO Module; the current task can also terminate its own 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 `Tconf` to specify one or more sets of application-wide hook functions that run whenever a task state changes in a particular way. For the TSK module, these functions are the `Create`, `Delete`, `Exit`, `Switch`, and `Ready` functions. The `HOOK` module adds an additional Initialization function.

A single set of hook functions can be specified for the TSK module itself. To create additional sets of hook functions, use the `HOOK` Module. When you create the first `HOOK` object, any TSK module hook functions you have specified are automatically placed in a `HOOK` object called `HOOK_KNL`. To set any properties of this object other than the Initialization function, use the TSK module properties. To set the

Initialization function property of the HOOK\_KNL object, use the HOOK object properties. If you configure only a single set of hook functions using the TSK module, the HOOK module is not used.

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 is 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 SWI handler. See Appendix A, Function Callability Table, 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 in the TSK Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

- ❑ **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 Tconf 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.

Tconf Name: ENABLETSK Type: Bool

Example: `bios.TSK.ENABLETSK = true;`

- ❑ **Object Memory.** The memory segment that contains the TSK objects created with Tconf.

Tconf Name: OBJMEMSEG Type: Reference

Example: `bios.TSK.OBJMEMSEG = prog.get("myMEM");`

- ❑ **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 Tconf or TSK\_create. The estimated minimum task size is shown in the status bar of the DSP/BIOS Configuration Tool. This property applies to TSK objects created both with Tconf and with TSK\_create.

Tconf Name: STACKSIZE Type: Int16

Example: `bios.TSK.STACKSIZE = 1024;`

- ❑ **Stack segment for dynamic tasks.** The default memory segment to contain task stacks 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.

Tconf Name: STACKSEG Type: Reference

Example: `bios.TSK.STACKSEG = prog.get("myMEM");`

- ❑ **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 Tconf and with TSK\_create.

Tconf Name: PRIORITY Type: EnumInt

Options: 1 to 15

Example: `bios.TSK.PRIORITY = 1;`

- ❑ **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.

Tconf Name: DRIVETSKTICK

Type: EnumString

Options: "PRD", "User"

Example: `bios.TSK.DRIVETSKTICK = "PRD";`

- ❑ **Create function.** The name of a function to call when any task is created. This includes tasks that are created statically and those created dynamically using TSK\_create. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally. The TSK\_create topic describes the Create function.

Tconf Name: CREATEFXN

Type: Extern

Example: `bios.TSK.CREATEFXN =  
prog.extern("tskCreate");`

- ❑ **Delete function.** The name of a function to call when any task is deleted at run-time with TSK\_delete. The TSK\_delete topic describes the Delete function.

Tconf Name: DELETEDFXN

Type: Extern

Example: `bios.TSK.DELETEDFXN =  
prog.extern("tskDelete");`

- ❑ **Exit function.** The name of a function to call when any task exits. The TSK\_exit topic describes the Exit function.

Tconf Name: EXITFXN

Type: Extern

Example: `bios.TSK.EXITFXN =  
prog.extern("tskExit");`

- ❑ **Call switch function.** Check this box if you want a function to be called when any task switch occurs.

Tconf Name: CALLSWITCFXN

Type: Bool

Example: `bios.TSK.CALLSWITCFXN = false;`

- ❑ **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. The TSK Module topic describes the Switch function.

Tconf Name: SWITCHFXN

Type: Extern

Example: `bios.TSK.SWITCHFXN =  
prog.extern("tskSwitch");`



- ☐ **Call ready function.** Check this box if you want a function to be called when any task becomes ready to run.

Tconf Name: CALLREADYFXN

Type: Bool

Example: `bios.TSK.CALLREADYFXN = false;`

- ☐ **Ready function.** The name of a function to call when any task becomes ready to run. The TSK Module topic describes the Ready function.

Tconf Name: READYFXN

Type: Extern

Example: `bios.TSK.READYFXN =  
prog.extern("tskReady");`

## TSK Object Properties

To create a TSK object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myTsk = bios.TSK.create("myTsk");
```

The following properties can be set for a TSK object in the TSK Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

### General tab

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

Tconf Name: comment

Type: String

Example: `myTsk.comment = "my TSK";`

- ☐ **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.

Tconf Name: autoAllocateStack

Type: Bool

Example: `myTsk.autoAllocateStack = true;`

- ☐ **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.

Tconf Name: manualStack

Type: Extern

Example: `myTsk.manualStack =  
prog.extern("myStack");`

- ☐ **Stack size.** Enter the size (in MADUs) of the stack space to allocate for this task. You must enter the size whether the application allocates the stack manually or automatically. 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.

Tconf Name: `stackSize`

Type: `Int16`

Example: `myTsk.stackSize = 1024;`

- ❑ **Stack Memory Segment.** If you set the "Automatically allocate stack" property to true, specify the memory segment to contain the stack space for this task.

Tconf Name: `stackMemSeg`

Type: `Reference`

Example: `myTsk.stackMemSeg = prog.get("myMEM");`

- ❑ **Priority.** The priority level for this task. A priority of -1 causes a task to be suspended until its priority is raised programmatically.

Tconf Name: `priority`

Type: `EnumInt`

Options: `-1, 0, 1 to 15`

Example: `myTsk.priority = 1;`

## Function tab

- ❑ **Task function.** The function to be executed when the task runs. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. (The DSP/BIOS Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.) If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally. If you compile C programs with the `-pm` or `-op2` options, you should precede C functions called by task threads with the `FUNC_EXT_CALLED` pragma. See the online help for the C compiler for details.

Tconf Name: `fxn`

Type: `Extern`

Example: `myTsk.fxn = prog.extern("tskFxn");`

- ❑ **Task function argument 0-7.** The arguments to pass to the task function. Arguments can be integers or labels.

Tconf Name: `arg0 to arg7`

Type: `Arg`

Example: `myTsk.arg0 = 0;`

## Advanced tab

- ❑ **Environment pointer.** A pointer to a globally-defined data structure this task can access. The task can get and set the task environment pointer with the `TSK_getenv` and `TSK_setenv` functions. If your program uses multiple `HOOK` objects, `HOOK_setenv` allows you to set individual environment pointers for each `HOOK` and `TSK` object combination.

Tconf Name: `envPointer`

Type: `Arg`

Example: `myTsk.envPointer = 0;`

- ☐ **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.

Tconf Name: `exitFlag`

Type: Bool

Example: `myTsk.exitFlag = true;`

- ☐ **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.

Tconf Name: `allocateTaskName`

Type: Bool

Example: `myTsk.allocateTaskName = false;`

- ☐ **order.** Set this property for all TSK objects so that the numbers match the sequence in which TSK functions with the same priority level should be executed.

Tconf Name: `order`

Type: Int16

Example: `myTsk.order = 2;`

**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><br><code>TSK_Handle newtask; /* handle of task switched to */</code> |
| <b>Return Value</b> | Void                                                                                                                                     |

**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 TSK\_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.

TSK\_checkstacks requires that the stack was initialized by DSP/BIOS. For dynamically-created tasks, initialization is controlled by the initstackflag attribute in the TSK\_Attrs structure passed to TSK\_create. Statically configured tasks always initialize the 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;      /* pointer to task function */
TSK_Attrs *attrs;   /* pointer to task attributes */
Arg      arg;      /* task arguments */
```

**Return Value**

```
TSK_Handle task;    /* task object handle */
```

**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).

The fxn parameter uses the Fxn type to pass a pointer to the function the TSK object should run. For example, if myFxn is a function in your program, you can create a TSK object to call that function as follows:

```
task = TSK_create((Fxn)myFxn, NULL);
```

You can use Tconf to specify an application-wide Create function that runs whenever a task is created. This includes tasks that are created statically and those created dynamically using TSK\_create. The default Create function is a no-op function.

For TSK objects created statically, the Create function is called during the BIOS\_start portion of the program startup process, which runs after the main() function and before the program drops into the idle loop.

For TSK objects created dynamically, 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 the Create function. The Create function declaration should be similar to this:

```
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, which is defined as follows.

```
struct TSK_Attrs { /* task attributes */
    Int    priority; /* execution priority */
    Ptr    stack;    /* pre-allocated stack */
    size_t stacksize; /* stack size in MADUs */
    Int    stackseg; /* mem seg for stack alloc */
    Ptr    environ;  /* global environ data struct */
    String name;     /* printable name */
    Bool   exitflag; /* prog termination requires */
                    /* this task to terminate */
    Bool   initstackflag; /* initialize task stack? */
};
```

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, the task is barred from execution until its priority is raised at a later time by TSK\_setpri. 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. 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.

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 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 the task must terminate before the program as a whole can terminate; this attribute defaults to `TRUE`.

The `initstackflag` attribute specifies whether the task stack is initialized to enable stack depth checking by `TSK_checkstacks`. This attribute applies both in cases where the stack attribute is `NULL` (stack is allocated by `TSK_create`) and where the stack attribute is used to specify a pre-allocated stack. If your application does not call `TSK_checkstacks`, you can reduce the time consumed by `TSK_create` by setting this attribute to `FALSE`.

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 an object's data structure. `MEM_alloc` must lock 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 2–240.

## Constraints and Calling Context

- ❑ `TSK_create` cannot be called from a 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.
- ❑ Task arguments passed to `TSK_create` cannot be greater than 32 bits in length; that is, 40-bit integers and Double or Long Double data types cannot be passed as arguments to the `TSK_create` function.
- ❑ You can reduce the size of your application program by creating objects with `Tconf` rather than using the `XXX_create` functions.

## See Also

`MEM_alloc`  
`SYS_error`  
`TSK_delete`  
`TSK_exit`

**TSK\_delete***Delete a task***C Interface**

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| <b>Syntax</b>       | <code>TSK_delete(task);</code>                                |
| <b>Parameters</b>   | <code>TSK_Handle task;</code> <i>/* task object handle */</i> |
| <b>Return Value</b> | <code>Void</code>                                             |

**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 Tconf 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 a 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 Tconf, SYS\_error is called.

**See Also**

MEM\_free  
TSK\_create



**TSK\_deltatime***Update task statistics with time difference***C Interface**

|                     |                                                |
|---------------------|------------------------------------------------|
| <b>Syntax</b>       | TSK_deltatime(task);                           |
| <b>Parameters</b>   | TSK_Handle task;      /* task object handle */ |
| <b>Return Value</b> | Void                                           |

**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 is never 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**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 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 Module semaphores should be used for mutual exclusion when possible.

**Constraints and Calling Context**

- ❑ Do not call any function that can cause the current task to block or otherwise affect the state of the scheduler within a TSK\_disable/TSK\_enable block. For example, SEM\_pend (if timeout is non-zero), TSK\_sleep, TSK\_yield, and MEM\_alloc can all cause blocking. Similarly, any MEM module call and any call that dynamically creates or deletes an object (XXX\_create or XXX\_delete) can affect the state of the scheduler. For a complete list, see the "Possible Context Switch" column in Section A.1, *Function Callability Table*.
- ❑ TSK\_disable cannot be called from a SWI or HWI.
- ❑ TSK\_disable cannot be called from the program's main() function.
- ❑ Do not call TSK\_enable when TSKs are already enabled. If you do so, the subsequent call to TSK\_disable will not disable TSK processing.

**See Also**

SEM Module  
TSK\_enable

**TSK\_enable***Enable DSP/BIOS task scheduler***C Interface****Syntax** TSK\_enable();**Parameters** Void**Return Value** Void**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**

- ❑ Do not call any function that can cause the current task to block or otherwise affect the state of the scheduler within a TSK\_disable/TSK\_enable block. For example, SEM\_pend (if timeout is non-zero), TSK\_sleep, TSK\_yield, and MEM\_alloc can all cause blocking. Similarly, any MEM module call and any call that dynamically creates or deletes an object (XXX\_create or XXX\_delete) can affect the state of the scheduler. For a complete list, see the "Possible Context Switch" column in Section A.1, *Function Callability Table*.
- ❑ TSK\_enable cannot be called from a SWI or HWI.
- ❑ TSK\_enable cannot be called from the program's main() function.
- ❑ Do not call TSK\_enable when TSKs are already enabled. If you do so, the subsequent call to TSK\_disable will not disable TSK processing.

**See Also**

SEM Module  
TSK\_disable

**TSK\_exit***Terminate execution of the current task***C Interface****Syntax**                      TSK\_exit();**Parameters**                  Void**Return Value**                Void**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 Tconf 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 an Exit function. Calling TSK\_self within an Exit function returns the task being exited. Your Exit function declaration should be similar to the following:

```
Void myExitFxn(Void);
```

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 a 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**

|                     |                                                                                  |
|---------------------|----------------------------------------------------------------------------------|
| <b>Syntax</b>       | <code>environ = TSK_getenv(task);</code>                                         |
| <b>Parameters</b>   | <code>TSK_Handle task;</code> <i>/* task object handle */</i>                    |
| <b>Return Value</b> | <code>Ptr                  environ;</code> <i>/* task environment pointer */</i> |

**Description**

TSK\_getenv returns the environment pointer of the specified task. The environment pointer, environ, references an arbitrary application-defined data structure.

If your program uses multiple HOOK objects, HOOK\_getenv allows you to get environment pointers you have set for a particular HOOK and TSK object combination.

**See Also**

HOOK\_getenv  
HOOK\_setenv  
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>  |

**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**

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| <b>Syntax</b>       | <code>name = TSK_getname(task);</code>                        |
| <b>Parameters</b>   | <code>TSK_Handle task;</code> <i>/* task object handle */</i> |
| <b>Return Value</b> | <code>String name;</code> <i>/* task name */</i>              |

**Description**     TSK\_getname returns the task's name.

For tasks created with Tconf, the name is available to this function only if the "Allocate Task Name on Target" property is set to true 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>       | <code>priority = TSK_getpri(task);</code>                          |
| <b>Parameters</b>   | <code>TSK_Handle task;</code> <i>/* task object handle */</i>      |
| <b>Return Value</b> | <code>Int</code> <code>priority;</code> <i>/* task priority */</i> |
| <b>Description</b>  | TSK_getpri returns the priority of task.                           |
| <b>See Also</b>     | TSK_setenv<br>TSK_seterr<br>TSK_setpri                             |

**TSK\_getsts***Get the handle of the task's STS object***C Interface****Syntax**`sts = TSK_getsts(task);`**Parameters**`TSK_Handle task;      /* task object handle */`**Return Value**`STS_Handle sts;      /* statistics object handle */`**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\_isTSK***Check to see if called in the context of a TSK***C Interface****Syntax**`result = TSK_isTSK(Void);`**Parameters**

Void

**Return Value**

Bool            result;     /\* TRUE if in TSK context, FALSE otherwise \*/

**Reentrant**

yes

**Description**

This macro returns TRUE when it is called within the context of a TSK or IDL function. It returns FALSE in all other contexts.

TSK\_isTSK() API returns TRUE when the current thread is neither a HWI nor a SWI. Thus, TSK\_isTSK() returns TRUE when it is invoked within a task thread, main(), or a task switch hook.

In previous versions of DSP/BIOS, calling the context checking functions from main() resulted in TRUE for HWI\_isHWI(). And, calling the context checking functions from a task switch hook resulted in TRUE for SWI\_isSWI(). This is no longer the case; they are identified as part of the TSK context.

In applications that contain no task threads, TSK\_isTSK() now returns TRUE from main() and from the IDL threads.

**See Also**

HWI\_isHWI  
SWI\_isSWI

**TSK\_itick***Advance the system alarm clock (interrupt use only)***C Interface****Syntax** TSK\_itick();**Parameters** Void**Return Value** Void

**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, 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 \*/**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 */<br>Ptr                  environ;   /* task environment pointer */ |
| <b>Return Value</b> | Void                                                                                                             |

**Description**                  TSK\_setenv sets the task environment pointer to environ. The environment pointer, environ, references an arbitrary application-defined data structure.

If your program uses multiple HOOK objects, HOOK\_setenv allows you to set individual environment pointers for each HOOK and TSK object combination.

|                 |                                                        |
|-----------------|--------------------------------------------------------|
| <b>See Also</b> | HOOK_getenv<br>HOOK_setenv<br>TSK_getenv<br>TSK_geterr |
|-----------------|--------------------------------------------------------|

**TSK\_seterr***Set task error number***C Interface**

|                     |                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | TSK_seterr(task, errno);                                                                    |
| <b>Parameters</b>   | TSK_Handle task;     /* task object handle */<br>Int            errno;   /* error number */ |
| <b>Return Value</b> | Void                                                                                        |

**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****Syntax**

```
oldpri = TSK_setpri(task, newpri);
```

**Parameters**

```
TSK_Handle task;    /* task object handle */  
Int         newpri; /* task's new priority */
```

**Return Value**

```
Int         oldpri; /* task's old priority */
```

**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, the 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 the 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                                          |

**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 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                                                                  |

**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 a 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

**Description**

TSK\_stat retrieves attribute values and status information about a task.

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 stack pointer */  
    size_t     used;  /* task stack used */  
};
```

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 runs when the preemption ends.

The current task can inquire about itself by passing the output of TSK\_self as the first argument to TSK\_stat. However, the task stack pointer (sp) in the TSK\_Stat structure is the value from the previous context switch.

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**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 HWI 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, 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> |

**Description**

TSK\_time returns the current value of the system alarm clock.

Note that since the system clock is usually updated asynchronously 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**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.

Tasks of higher priority preempt the currently running task without the need for a call to TSK\_yield. If only lower-priority tasks are ready to run when you call TSK\_yield, the current task continues to run. Control does not pass to a lower-priority task.

**Constraints and  
Calling Context**

- ❑ When called within an HWI, 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.

**See Also**

TSK\_sleep

## 2.33 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.** Free memory.
- ❑ **\*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.



**Note: RTS Functions Callable from TSK Threads Only**

Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK\_pend and LCK\_post. As a result, RTS functions that call LCK\_pend or LCK\_post *must not be called in the context of a SWI or HWI thread*. For a list of RTS functions that should not be called from a SWI or an HWI function, see “LCK\_pend” on page 2-210.

To determine whether a particular RTS function uses LCK\_pend, refer to the source code for that function shipped with Code Composer Studio. The following table shows some of the RTS functions that call LCK\_pend in certain versions of Code Composer Studio:

|         |          |          |          |
|---------|----------|----------|----------|
| fprintf | printf   | vfprintf | sprintf  |
| vprintf | vsprintf | clock    | strftime |
| minit   | malloc   | realloc  | free     |
| calloc  | rand     | srand    | getenv   |

The C++ new operator calls malloc, which in turn calls LCK\_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

---

# Function Callability and Error Tables

---

---

---

This appendix provides tables describing TMS320C6000 errors and function callability.

| Topic                                    | Page |
|------------------------------------------|------|
| A.1 Function Callability Table . . . . . | A-2  |
| A.2 DSP/BIOS Error Codes . . . . .       | A-12 |

## A.1 Function Callability Table

The following table indicates what types of threads can call each of the DSP/BIOS functions. The Possible Context Switch column indicates whether another thread may be run as a result of this function. For example, the function may block on a resource or it may make another thread ready to run. The Possible Context Switch column does not indicate whether the function disables interrupts that might schedule higher-priority threads.

*Table A-1 Function Callability*

| Function         | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| ATM_andi         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_andu         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_cleari       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_clearu       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_deci         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_decu         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_inci         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_incu         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_ori          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_oru          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_seti         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ATM_setu         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_getMar    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_getMode   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_getSize   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_inv       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_invL1pAll | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_setMar    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_setMode   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_setSize   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_wait      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_wb        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_wbAll     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_wbInv     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BCACHE_wbInvAll  | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BUF_alloc        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function              | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|-----------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| BUF_create            | Yes                  | No                   | No                   | Yes                            | Yes                         |
| BUF_delete            | Yes                  | No                   | No                   | Yes                            | Yes                         |
| BUF_free              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| BUF_maxbuff           | Yes                  | No                   | No                   | No                             | Yes                         |
| BUF_stat              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C62_disableIER        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C62_enableIER         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C62_plug              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C64_disableIER        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C64_enableIER         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| C64_plug              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_countspms         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_cpuCyclesPerHtime | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_cpuCyclesPerLtime | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_gethtime          | Yes                  | Yes                  | Yes                  | No                             | No                          |
| CLK_getltime          | Yes                  | Yes                  | Yes                  | No                             | No                          |
| CLK_getprd            | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_reconfig          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| CLK_start             | Yes                  | Yes                  | Yes                  | No                             | No                          |
| CLK_stop              | Yes                  | Yes                  | Yes                  | No                             | No                          |
| DEV_createDevice      | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| DEV_deleteDevice      | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| DEV_match             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ECM_disableEvent      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ECM_dispatch          | No                   | No                   | Yes                  | No                             | No                          |
| ECM_dispatchPlug      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| ECM_enableEvent       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| EXC_clearLastStatus   | Yes                  | Yes                  | Yes*                 | No                             | Yes                         |
| EXC_dispatch          | No                   | No                   | Yes                  | No                             | No                          |
| EXC_exceptionHandler  | No                   | No                   | No*                  | No                             | No                          |
| EXC_exceptionHook     | No                   | No                   | No*                  | No                             | No                          |
| EXC_external          | No                   | No                   | No*                  | No                             | No                          |
| EXC_externalHook      | No                   | No                   | No*                  | No                             | No                          |
| EXC_evtEvtClear       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| EXC_evtExpEnable      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function            | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|---------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| EXC_getLastStatus   | Yes                  | Yes                  | Yes*                 | No                             | Yes                         |
| EXC_internal        | No                   | No                   | No*                  | No                             | No                          |
| EXC_internalHook    | No                   | No                   | No*                  | No                             | No                          |
| EXC_nmi             | No                   | No                   | No*                  | No                             | No                          |
| EXC_nmiHook         | No                   | No                   | No*                  | No                             | No                          |
| GBL_getClkin        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| GBL_getFrequency    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| GBL_getProcid       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| GBL_getVersion      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| GBL_setFrequency    | No                   | No                   | No                   | No                             | Yes                         |
| GBL_setProcid       | No                   | No                   | No                   | No                             | No*                         |
| GIO_abort           | Yes                  | No*                  | No*                  | Yes                            | No                          |
| GIO_control         | Yes                  | No*                  | No*                  | Yes                            | Yes                         |
| GIO_create          | Yes                  | No                   | No                   | No                             | Yes                         |
| GIO_delete          | Yes                  | No                   | No                   | Yes                            | Yes                         |
| GIO_flush           | Yes                  | No*                  | No*                  | Yes                            | No                          |
| GIO_new             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| GIO_read            | Yes                  | No*                  | No*                  | Yes                            | Yes*                        |
| GIO_submit          | Yes                  | Yes*                 | Yes*                 | Yes                            | Yes*                        |
| GIO_write           | Yes                  | No*                  | No*                  | Yes                            | Yes*                        |
| HOOK_getenv         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HOOK_setenv         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HST_getpipe         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_applyWugenMasks | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_disable         | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_disableWugen    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_dispatchPlug    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_enable          | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| HWI_enableWugen     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_enter           | No                   | No                   | Yes                  | No                             | No                          |
| HWI_eventMap        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_exit            | No                   | No                   | Yes                  | Yes                            | No                          |
| HWI_getWugenMasks   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_ierToWugenMasks | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| HWI_isHWI           | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function              | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|-----------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| HWI_restore           | Yes                  | Yes                  | Yes                  | Yes*                           | Yes                         |
| IDL_run               | Yes                  | No                   | No                   | No                             | No                          |
| LCK_create            | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| LCK_delete            | Yes                  | No                   | No                   | Yes*                           | No                          |
| LCK_pend              | Yes                  | No                   | No                   | Yes*                           | No                          |
| LCK_post              | Yes                  | No                   | No                   | Yes*                           | No                          |
| LOG_disable           | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_enable            | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_error             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_event             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_event5            | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_message           | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_printf            | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_printf4           | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| LOG_reset             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MBX_create            | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MBX_delete            | Yes                  | No                   | No                   | Yes*                           | No                          |
| MBX_pend              | Yes                  | Yes*                 | Yes*                 | Yes*                           | No                          |
| MBX_post              | Yes                  | Yes*                 | Yes*                 | Yes*                           | Yes*                        |
| MEM_alloc             | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_calloc            | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_define            | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_free              | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_getBaseAddress    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MEM_increaseTableSize | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_redefine          | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_stat              | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_undefine          | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MEM_valloc            | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| MPC_getPA             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MPC_getPageSize       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MPC_getPrivMode       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MPC_setBufferPA       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MPC_setPA             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MPC_setPrivMode       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function               | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|------------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| _MPC_getLastMPFAR      | No                   | No                   | No*                  | No                             | No                          |
| _MPC_getLastMPFSR      | No                   | No                   | No*                  | No                             | No                          |
| _MPC_exceptionHandler  | No                   | No                   | No*                  | No                             | No                          |
| _MPC_externalHandler   | No                   | No                   | No*                  | No                             | No                          |
| _MPC_internalHandler   | No                   | No                   | No*                  | No                             | No                          |
| _MPC_userHook          | No                   | No                   | No*                  | No                             | No                          |
| MSGQ_alloc             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_close             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_count             | Yes                  | Yes*                 | Yes*                 | No                             | No                          |
| MSGQ_free              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_get               | Yes                  | Yes*                 | Yes*                 | Yes*                           | No                          |
| MSGQ_getAttrs          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_getDstQueue       | Yes                  | Yes                  | Yes                  | No                             | No                          |
| MSGQ_getMsgId          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_getMsgSize        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_getSrcQueue       | Yes                  | Yes                  | Yes                  | No                             | No                          |
| MSGQ_isLocalQueue      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_locate            | Yes                  | No                   | No                   | Yes                            | No                          |
| MSGQ_locateAsync       | Yes                  | Yes                  | Yes                  | No                             | No                          |
| MSGQ_open              | Yes                  | Yes*                 | Yes*                 | Yes*                           | Yes                         |
| MSGQ_put               | Yes                  | Yes                  | Yes                  | No                             | No                          |
| MSGQ_release           | Yes                  | Yes                  | Yes                  | No                             | No                          |
| MSGQ_setErrorHandler   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_setMsgId          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| MSGQ_setSrcQueue       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_alloc              | Yes                  | Yes                  | Yes                  | Yes                            | Yes                         |
| PIP_free               | Yes                  | Yes                  | Yes                  | Yes                            | Yes                         |
| PIP_get                | Yes                  | Yes                  | Yes                  | Yes                            | Yes                         |
| PIP_getReaderAddr      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_getReaderNumFrames | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_getReaderSize      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_getWriterAddr      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_getWriterNumFrames | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_getWriterSize      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PIP_peek               | Yes                  | Yes                  | Yes                  | No                             | Yes                         |



| Function                    | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|-----------------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| PIP_put                     | Yes                  | Yes                  | Yes                  | Yes                            | Yes                         |
| PIP_reset                   | Yes                  | Yes                  | Yes                  | Yes                            | Yes                         |
| PIP_setWriterSize           | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PRD_getticks                | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PRD_start                   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PRD_stop                    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PRD_tick                    | Yes                  | Yes                  | Yes                  | Yes                            | No                          |
| PWRM_changeSetpoint         | Yes                  | Yes*                 | No                   | No                             | No                          |
| PWRM_configure              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getCapabilities        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getConstraintInfo      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getCPULoad             | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getCurrentSetpoint     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getDependencyCount     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getLoadMonitorInfo     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getNumSetpoints        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getSetpointInfo        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_getTransitionLatency   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_registerConstraint     | Yes                  | No                   | No                   | Yes                            | Yes                         |
| PWRM_registerNotify         | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| PWRM_releaseDependency      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_resetCPULoadHistory    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_setDependency          | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_signalEvent            | Yes                  | Yes*                 | Yes*                 | Yes                            | No                          |
| PWRM_sleepDSP               | Yes                  | Yes*                 | No                   | No                             | No                          |
| PWRM_startCPULoadMonitoring | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_stopCPULoadMonitoring  | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_unregisterConstraint   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| PWRM_unregisterNotify       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_create                  | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| QUE_delete                  | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| QUE_dequeue                 | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_empty                   | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_enqueue                 | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_get                     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function                 | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main(?) |
|--------------------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| QUE_head                 | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_insert               | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_new                  | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_next                 | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_prev                 | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_put                  | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| QUE_remove               | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| RTDX_channelBusy         | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_CreateInputChannel  | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_CreateOutputChannel | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_disableInput        | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_disableOutput       | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_enableInput         | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_enableOutput        | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_isInputEnabled      | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_isOutputEnabled     | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_read                | Yes                  | Yes                  | No                   | No                             | No                          |
| RTDX_readNB              | Yes                  | Yes                  | No                   | No                             | No                          |
| RTDX_sizeofInput         | Yes                  | Yes                  | No                   | No                             | Yes                         |
| RTDX_write               | Yes                  | Yes                  | No                   | No                             | No                          |
| SEM_count                | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SEM_create               | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| SEM_delete               | Yes                  | Yes*                 | No                   | Yes*                           | No                          |
| SEM_new                  | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SEM_pend                 | Yes                  | Yes*                 | Yes*                 | Yes*                           | No                          |
| SEM_pendBinary           | Yes                  | Yes*                 | Yes*                 | Yes*                           | No                          |
| SEM_post                 | Yes                  | Yes                  | Yes                  | Yes*                           | Yes                         |
| SEM_postBinary           | Yes                  | Yes                  | Yes                  | Yes*                           | Yes                         |
| SEM_reset                | Yes                  | No                   | No                   | No                             | Yes                         |
| SIO_bufsize              | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SIO_create               | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| SIO_ctrl                 | Yes                  | Yes                  | No                   | No                             | Yes                         |
| SIO_delete               | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| SIO_flush                | Yes                  | Yes*                 | No                   | No                             | No                          |
| SIO_get                  | Yes                  | No                   | No                   | Yes*                           | Yes*                        |

| Function       | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main(?) |
|----------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| SIO_idle       | Yes                  | Yes*                 | No                   | Yes*                           | No                          |
| SIO_issue      | Yes                  | Yes                  | No                   | No                             | Yes                         |
| SIO_put        | Yes                  | No                   | No                   | Yes*                           | Yes*                        |
| SIO_ready      | Yes                  | Yes                  | Yes                  | No                             | No                          |
| SIO_reclaim    | Yes                  | Yes*                 | No                   | Yes*                           | Yes*                        |
| SIO_reclaimx   | Yes                  | Yes*                 | No                   | Yes*                           | Yes*                        |
| SIO_segid      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SIO_select     | Yes                  | Yes*                 | No                   | Yes*                           | No                          |
| SIO_staticbuf  | Yes                  | Yes                  | No                   | No                             | Yes                         |
| STS_add        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| STS_delta      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| STS_reset      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| STS_set        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SWI_andn       | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_andnHook   | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_create     | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| SWI_dec        | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_delete     | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| SWI_disable    | Yes                  | Yes                  | No                   | No                             | No                          |
| SWI_enable     | Yes                  | Yes                  | No                   | Yes*                           | No                          |
| SWI_getattr    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SWI_getmbox    | No                   | Yes                  | No                   | No                             | No                          |
| SWI_getpri     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SWI_inc        | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_isSWI      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SWI_or         | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_orHook     | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_post       | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| SWI_raisepri   | No                   | Yes                  | No                   | No                             | No                          |
| SWI_restorepri | No                   | Yes                  | No                   | Yes                            | No                          |
| SWI_self       | No                   | Yes                  | No                   | No                             | No                          |
| SWI_setattr    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_abort      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_atexit     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_error      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |

| Function        | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? | Callable<br>from<br>main()? |
|-----------------|----------------------|----------------------|----------------------|--------------------------------|-----------------------------|
| SYS_exit        | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_printf      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_putchar     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_sprintf     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_vprintf     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| SYS_vsprintf    | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TRC_disable     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TRC_enable      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TRC_query       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_checkstacks | Yes                  | No                   | No                   | No                             | No                          |
| TSK_create      | Yes                  | No                   | No                   | Yes*                           | Yes                         |
| TSK_delete      | Yes                  | No                   | No                   | Yes*                           | No                          |
| TSK_deltatime   | Yes                  | Yes                  | Yes                  | No                             | No                          |
| TSK_disable     | Yes                  | No                   | No                   | No                             | No                          |
| TSK_enable      | Yes                  | No                   | No                   | Yes*                           | No                          |
| TSK_exit        | Yes                  | No                   | No                   | Yes*                           | No                          |
| TSK_getenv      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_geterr      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_getname     | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_getpri      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_getsts      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_isTSK       | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_itick       | No                   | Yes                  | Yes                  | Yes                            | No                          |
| TSK_self        | Yes                  | Yes                  | Yes                  | No                             | No                          |
| TSK_setenv      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_seterr      | Yes                  | Yes                  | Yes                  | No                             | Yes                         |
| TSK_setpri      | Yes                  | Yes                  | Yes                  | Yes*                           | Yes                         |
| TSK_settime     | Yes                  | Yes                  | Yes                  | No                             | No                          |
| TSK_sleep       | Yes                  | No                   | No                   | Yes*                           | No                          |
| TSK_stat        | Yes                  | Yes*                 | Yes*                 | No                             | Yes                         |
| TSK_tick        | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |
| TSK_time        | Yes                  | Yes                  | Yes                  | No                             | No                          |
| TSK_yield       | Yes                  | Yes                  | Yes                  | Yes*                           | No                          |

Note: \*See the appropriate API reference page for more information.

Table A-2 RTS Function Calls

| Function | Callable<br>by TSKs? | Callable<br>by SWIs? | Callable by<br>HWIs? | Possible<br>Context<br>Switch? |
|----------|----------------------|----------------------|----------------------|--------------------------------|
| calloc   | Yes                  | No                   | No                   | Yes*                           |
| clock    | Yes                  | No                   | No                   | Yes*                           |
| fprintf  | Yes                  | No                   | No                   | Yes*                           |
| free     | Yes                  | No                   | No                   | Yes*                           |
| getenv   | Yes                  | No                   | No                   | Yes*                           |
| malloc   | Yes                  | No                   | No                   | Yes*                           |
| minit    | Yes                  | No                   | No                   | Yes*                           |
| printf   | Yes                  | No                   | No                   | Yes*                           |
| rand     | Yes                  | No                   | No                   | Yes*                           |
| realloc  | Yes                  | No                   | No                   | Yes*                           |
| sprintf  | Yes                  | No                   | No                   | Yes*                           |
| srand    | Yes                  | No                   | No                   | Yes*                           |
| strftime | Yes                  | No                   | No                   | Yes*                           |
| vfprintf | Yes                  | No                   | No                   | Yes*                           |
| vprintf  | Yes                  | No                   | No                   | Yes*                           |
| vsprintf | Yes                  | No                   | No                   | Yes*                           |

Note: \*See section 2.33, *std.h* and *stdlib.h* functions, page 2-562 for more information.

## A.2 DSP/BIOS Error Codes

Table A-3 Error Codes

| Name          | Value | SYS_Errors[Value]                                                                                                                                                                                          |
|---------------|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SYS_OK        | 0     | "(SYS_OK)"                                                                                                                                                                                                 |
| SYS_EALLOC    | 1     | "(SYS_EALLOC): segid = %d, size = %u, align = %u"<br>Memory allocation error.                                                                                                                              |
| SYS_EFREE     | 2     | "(SYS_EFREE): segid = %d, ptr = 0x%x, size = %u"<br>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"<br>The device being opened is not configured into the system.                                                                                                             |
| SYS_EBUSY     | 4     | "(SYS_EBUSY): device in use"<br>The device is already opened by the maximum number of users.                                                                                                               |
| SYS_EINVAL    | 5     | "(SYS_EINVAL): invalid parameter"<br>An invalid parameter was passed.                                                                                                                                      |
| SYS_EBADIO    | 6     | "(SYS_EBADIO): device failure"<br>The device was unable to support the I/O operation.                                                                                                                      |
| SYS_EMODE     | 7     | "(SYS_EMODE): invalid mode"<br>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"<br>Used by SPOX-MATH when type of operation does not match vector or filter type.                                                                                            |
| SYS_ETIMEOUT  | 9     | "(SYS_ETIMEOUT): timeout error"<br>Used by device drivers to indicate that reclaim timed out.                                                                                                              |
| SYS_EEOF      | 10    | "(SYS_EEOF): end-of-file error"<br>Used by device drivers to indicate the end of a file.                                                                                                                   |
| SYS_EDEAD     | 11    | "(SYS_EDEAD): previously deleted object"<br>An attempt was made to use an object that has been deleted.                                                                                                    |
| SYS_EBADOBJ   | 12    | "(SYS_EBADOBJ): invalid object"<br>An attempt was made to use an object that does not exist.                                                                                                               |
| SYS_ENOTIMPL  | 13    | "(SYS_ENOTIMPL): action not implemented"<br>An attempt was made to use an action that is not implemented.                                                                                                  |
| SYS_ENOTFOUND | 14    | "(SYS_ENOTFOUND): resource not found"<br>An attempt was made to use a resource that could not be found.                                                                                                    |
| SYS_EUSER     | >=256 | "(SYS_EUSER): <user-defined string>"<br>User-defined error.                                                                                                                                                |

# C6000 DSP/BIOS Register Usage

---

---

---

This appendix provides tables describing the TMS320C6000™ register conventions in terms of preservation across multi-threaded context switching and preconditions.

| Topic                          | Page |
|--------------------------------|------|
| B.1 Overview.....              | B-2  |
| B.2 Register Conventions ..... | B-2  |

## B.1 Overview

In a multi-threaded application using DSP/BIOS, it is necessary to know which registers can or cannot be modified. Furthermore, users need to understand which registers need to be saved/restored across a function call or an interrupt.

The following definitions describe the various possible register handling behaviors:

- ❑ **Scratch register.** These registers are saved/restored by the HWI dispatcher or HWI\_enter/HWI\_exit with temporary register bit masks.
- ❑ **Preserved register.** These registers are saved/restored during a TSK context switch.
- ❑ **Initialized register.** These registers are set to a particular value during HWI processing and restored to their incoming value upon exiting to the interrupt routine.
- ❑ **Read-Only register.** These registers may be read but must not be modified.
- ❑ **Global register.** These registers are shared across all threads in the system. To make a temporary change, save the register, make the change, and then restore it.
- ❑ **Other.** These registers do not fit into one of the categories above.

## B.2 Register Conventions

*Table B-1 Register and Status Bit Handling*

| Register                        | Status Bit | Register or Status Bit Name | Type        | Notes                                    |
|---------------------------------|------------|-----------------------------|-------------|------------------------------------------|
| A0-A9,<br>B0-B9                 |            | General purpose registers   | Scratch     |                                          |
| A10-A12,<br>A14-A15,<br>B10-B13 |            | General purpose registers   | Preserved   |                                          |
| A13                             |            | Frame pointer               | Preserved   |                                          |
| B14                             |            | Data page pointer           | Initialized | HWI sets to bss before calling ISR       |
| B15                             |            | Stack pointer               | Initialized | HWI sets to HWI stack before calling ISR |



| Register                | Status Bit  | Register or Status Bit Name           | Type        | Notes                                                |
|-------------------------|-------------|---------------------------------------|-------------|------------------------------------------------------|
| A16-A31**,<br>B16-B31** |             | General purpose registers             | Scratch     |                                                      |
| AMR                     |             | Addressing mode register              | Initialized | HWI sets to 0 before calling ISR                     |
| CSR                     | GIE         | Global interrupt enable               | Global      |                                                      |
|                         | PGIE        | Previous global interrupt enable      | Global      |                                                      |
|                         | DCC         | Data cache control mode               | Preserved   |                                                      |
|                         | PCC         | Program cache control mode            | Preserved   |                                                      |
|                         | EN          | Endian bit                            | Read-Only   |                                                      |
|                         | SAT         | Saturation bit                        | Scratch     |                                                      |
|                         | PWRD        | Control power-down modes              | Global      |                                                      |
|                         | Revision ID | Revision ID                           | Read-Only   |                                                      |
|                         | CPU ID      | CPU ID                                | Read-Only   |                                                      |
| IFR                     |             | Interrupt flag register               | Read-Only   |                                                      |
| ISR                     |             | Interrupt set register                | Other       | Cannot be read                                       |
| ICR                     |             | Interrupt clear register              | Other       | Cannot be read                                       |
| IER                     |             | Interrupt enable register             | Read-Only   |                                                      |
| ISTP                    |             | Interrupt service table pointer       | Read-Only   |                                                      |
| IRP                     |             | Interrupt return pointer              | Global      | Can be modified with interrupts disabled.            |
| NRP                     |             | Non-maskable interrupt return pointer | Read-Only   |                                                      |
| PCE1                    |             | Program counter, E1 phase             | Read-Only   |                                                      |
| FADCR*                  | Rmode       | Rounding mode                         | Global      | Currently DSP/BIOS does not deal with this register. |
|                         | UNDER       | Underflow status bit                  |             |                                                      |
|                         | INEX        | Exponent status bit                   |             |                                                      |
|                         | OVER        | Overflow status bit                   |             |                                                      |
|                         | INFO        | Signed infinity status bit            |             |                                                      |
|                         | INVAL       | INVAL status bit                      |             |                                                      |

| Register | Status Bit | Register or Status Bit Name       | Type   | Notes                                                |
|----------|------------|-----------------------------------|--------|------------------------------------------------------|
| FAUCR*   | DEN2       | Denormalized number               | Global | Currently DSP/BIOS does not deal with this register. |
|          | DEN1       | Denormalized number               |        |                                                      |
|          | NAN2       | NaN number                        |        |                                                      |
|          | NAN1       | NaN number                        |        |                                                      |
|          | DIV0       | DIV0 status bit                   |        |                                                      |
|          | UNORD      | UNORD status bit                  |        |                                                      |
|          | UNDER      | Underflow status bit              |        |                                                      |
|          | INEX       | Exponent status bit               |        |                                                      |
|          | OVER       | Overflow status bit               |        |                                                      |
|          | INFO       | Signed infinity status bit        |        |                                                      |
|          | INVAL      | INVAL status bit                  |        |                                                      |
|          | DEN2       | Denormalized number               |        |                                                      |
|          | DEN1       | Denormalized number               |        |                                                      |
| FMCR*    | NAN2       | NaN number                        | Global | Currently DSP/BIOS does not deal with this register. |
|          | NAN1       | NaN number                        |        |                                                      |
|          | Rmode      | Rounding mode                     |        |                                                      |
|          | UNDER      | Underflow status bit              |        |                                                      |
|          | INEX       | Exponent status bit               |        |                                                      |
|          | OVER       | Overflow status bit               |        |                                                      |
|          | INFO       | Signed infinity status bit        |        |                                                      |
|          | INVAL      | INVAL status bit                  |        |                                                      |
|          | DEN2       | Denormalized number               |        |                                                      |
|          | DEN1       | Denormalized number               |        |                                                      |
|          | NAN2       | NaN number                        |        |                                                      |
|          | NAN1       | NaN number                        |        |                                                      |
|          |            |                                   |        |                                                      |
| GFPGFR** |            | Galois Field Polynomial Generator | Global | Currently DSP/BIOS does not deal with this register. |
|          |            |                                   |        |                                                      |

| Register | Status Bit | Register or Status Bit Name        | Type              | Notes                                                                                                                                                                                                                                                  |
|----------|------------|------------------------------------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TSR+     | GIE        | Global interrupt enable            | Global            | DSP/BIOS does not maintain this C64x+ status bit. Since DSP/BIOS does not do a "return from interrupt" for certain task switches, your application should not expect this bit to correctly indicate whether an interrupt is currently being processed. |
|          | SGIE       | Saved global interrupt enable      | Global            |                                                                                                                                                                                                                                                        |
|          | GEE        | Global exception enable            | Read-Only         |                                                                                                                                                                                                                                                        |
|          | XEN        | Maskable exception enable          | Read-Only         |                                                                                                                                                                                                                                                        |
|          | DBGM       | Emulator debug mask                | Read-Only         |                                                                                                                                                                                                                                                        |
|          | CXM        | Current execution mode             | Read-Only         |                                                                                                                                                                                                                                                        |
|          | INT        | Interrupt processing               | Read-Only / Other |                                                                                                                                                                                                                                                        |
|          | EXC        | Exception processing               | Read-Only         |                                                                                                                                                                                                                                                        |
|          | SPLX       | SPLOOP executing                   | Read-Only         |                                                                                                                                                                                                                                                        |
|          | IB         | Interrupt blocked                  | Read-Only         |                                                                                                                                                                                                                                                        |
| ITSR+    |            | Interrupt task state register      | Global            |                                                                                                                                                                                                                                                        |
| NTSR+    |            | NMI/Exception task state register  | Global            |                                                                                                                                                                                                                                                        |
| EFR+     |            | Exception flag register            | Read-Only         |                                                                                                                                                                                                                                                        |
| ECR+     |            | Exception clear register           | Read-Only         |                                                                                                                                                                                                                                                        |
| IERR+    |            | Internal exception cause register  | Read-Only         |                                                                                                                                                                                                                                                        |
| SSR+     |            | Saturation status register         | Global            |                                                                                                                                                                                                                                                        |
| ILC+     |            | Inner loop SPL buffer count        | Global            |                                                                                                                                                                                                                                                        |
| RILC+    |            | Reload inner loop SPL buffer count | Global            |                                                                                                                                                                                                                                                        |
| GPLYA+   |            | GMPY polynomial for A side         | Scratch, Preserve |                                                                                                                                                                                                                                                        |
| GPLYB+   |            | GMPY polynomial for B side         | Scratch, Preserve |                                                                                                                                                                                                                                                        |

| Register | Status Bit | Register or Status Bit Name            | Type      | Notes |
|----------|------------|----------------------------------------|-----------|-------|
| TSCL+    |            | Low half of 64-bit time stamp counter  | Read-Only |       |
| TSCH+    |            | High half of 64-bit time stamp counter | Read-Only |       |
| DNUM+    |            | DSP number                             | Read-Only |       |
| DIER+    |            | Debug interrupt enable register        | Global    |       |

**Notes:**

\* — Denotes registers available on the 'C67x, 'C67x+ to support floating point operations.

\*\* — Denotes registers available on the 'C64x, 'C67x+ only.

+ — Denotes registers available on the 'C64x+ only.

The General purpose registers follow the 'C' compiler conventions. IRP can be used as a scratch register only when interrupts are disabled. ITSR and NTSR are identical copies of TSR, see TSR for details on each individual status bit.

For the 'C67x FADCR, FAUCR, and FMCR registers, the compiler assumes the nearest rounding mode is used. This is assumed to be the default mode at power-up. The compiler does not actually do anything to set it up that way, nor does it ever write or read these registers. These registers are completely under user control. Code may generate slightly different results if you change these registers.

# C64x+ Exception Support

---

---

---

This appendix provides describes support for C64x+ exception handling.

| Topic                                   | Page |
|-----------------------------------------|------|
| C.1 C64x+ Exception Support.....        | C-2  |
| C.2 Using the DSP/BIOS EXC Module ..... | C-3  |
| C.3 Data Types and Macros .....         | C-6  |
| C.4 EXC Module .....                    | C-7  |
| C.5 _MPC Module.....                    | C-16 |

## C.1 C64x+ Exception Support

DSP/BIOS provides exception support for the C64x+ family of DSPs through the EXC module. This module provides various handler functions that print exception data to the system log. The handler functions also call "user hook functions" at key locations. You can write hook functions to extend the behavior of the EXC module. The EXC module also records exception information for later retrieval by user code.

You can use DSP/BIOS exception support as-is; it provides useful diagnostic information. You can also extend it or replace it altogether.

A key point is that an exception indicates a fatal error. Exception processing should not attempt to return to the code that was interrupted. Exception processing is essentially a dead-end for the system, and should be limited to retrieving diagnostic information and/or shutting down the system. DSP/BIOS exception support is based on this idea. As soon as an exception is called, the "context" of HWI, SWI, and task threads no longer exists. (If you want to extend exception handling to include recovering from exceptions, you can write your own version of EXC\_dispatch and use the HWI dispatcher or HWI\_enter/HWI\_exit to maintain "context" within exception handling.)

### C.1.1 About C64x+ Exceptions

Exceptions are situations that trigger the NMI interrupt. The types of exceptions are:

- ❑ **Software-generated exceptions.** System calls that generate a SWE instruction are treated as exceptions. EXC\_dispatch calls an internal function to handle SWE exceptions.
- ❑ **External exceptions.** The C64x+ has a set of 128 system events. These events can be routed to interrupts and handled by the DSP/BIOS HWI and ECM modules. Alternatively, system events can be routed to the exception combiner (whose output goes only to the NMI pin). To cause an event to generate an exception, you must enable it—for example, by calling EXC\_evtExpEnable. The EXC module doesn't enable exception generation for any individual system event.

The EXC module handles external exceptions by routing them to EXC\_exceptionHandler, which calls the EXC\_external API. This API simply reports that an external exception occurred unless you write a hook function to provides more detail about an exception type.

- ❑ **Internal exceptions.** These are handled directly by the CPU. They are not related to events as are external exceptions. There is a CPU register (IERR) to report information about them. These exceptions

are routed to `EXC_exceptionHandler`, which calls the `EXC_internal` API to handle them.

- ❑ **Legacy NMI.** These are routed to `EXC_exceptionHandler`, which calls the `EXC_nmi` API to handle them.

## C.2 Using the DSP/BIOS EXC Module

This section provides a general description and general usage guidelines for DSP/BIOS C64x+ exception support. For further details, see Section C.4, *EXC Module*.

Source code for the EXC module is provided in the `src/exc` subdirectory of the DSP/BIOS installation.

The EXC module initializes DSP/BIOS to respond to C64x+ exceptions. It does this by enabling the GEE and XEN bits in the TSR register, and then installing an exception handler in the NMI vector. Once enabled, GEE cannot be disabled without resetting the CPU. This initialization allows internal exceptions to be recognized and routed to the NMI handler and then processed by `EXC_exceptionHandler`.

See the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for information about exception-related registers.

### C.2.1 Enabling and Disabling EXC Support

By default, the EXC module is enabled. To disable it, set the “Enable EXC module exception processing” field in the HWI Manager Properties to false. You can also disable the EXC module in a Tconf script with the following statement:

```
bios.HWI.ENABLEEXC = false;
```

Note that the EXC module does not have its own “module” in the configuration tools. It is, however, a module in the DSP/BIOS kernel. The HWI module is simply used as a container for the single EXC configuration property.

When enabled, EXC support configures the `HWI_NMI` object to run the `EXC_dispatch` function. You may alternately choose to configure your own function instead using the configuration tool. Source code for `EXC_dispatch` is provided in the `src/exc` subdirectory of the DSP/BIOS installation; you can use this as a starting point or an example.

If you use any EXC module APIs in your application source code, add the following line to your source file:

```
#include <exc.h>
```

### C.2.2 Out-of-the-Box EXC Behavior

The EXC module prints messages to the system log named LOG\_system. This log's output can be observed in CCStudio in a LOG window named "Execution Graph Details". These messages are intermixed with details of standard DSP/BIOS scheduling events and are flagged in the Execution Graph itself with a blue box.

After all processing is finished, EXC calls SYS\_abort to terminate the system. In general, when an application lands in SYS\_abort, you should look in the "Execution Graph Details" window for a message related to the abort.

By default, EXC processes only internal exceptions and legacy NMI occurrences. It also prints and records general exception information, such as the NRP (which points to the area where the exception occurred). If you enable any external events to generate exceptions, EXC doesn't report those exceptions. You will need to create a hook function that reports details about particular external events you enable.

### C.2.3 Extending EXC Exception Processing

There are four EXC function hooks you can use to gain processing control during exception processing. These hooks are:

- ❑ EXC\_exceptionHook
- ❑ EXC\_externalHook
- ❑ EXC\_internalHook
- ❑ EXC\_nmiHook

See Section C.4, *EXC Module* for details.

In addition, the \_MPC APIs provide a \_MPC\_userHook hook.

If you want to further customize EXC module behavior, source code for EXC\_dispatch is provided in the exc\_asm.s64P file in the src/exc subdirectory of the DSP/BIOS installation. Source code for other EXC functions is in the exc.c file in the same subdirectory.



## C.2.4 Interactions with the MPC Module

The DSP/BIOS MPC Module supports the C64x+ Memory Protection Controllers. The MPC hardware generates exceptions when an access that violates permissions occurs. The DSP/BIOS MPC module contains code that reports permission violations.

See section 2.18, *MPC Module*, page 2-260 for information about enabling the MPC module and using its APIs.

When enabled, the MPC module assigns its exception handling functions (with a prefix of “\_MPC”) to the EXC exception handling hooks.

The MPC module enables and handles only MPC-related events. Specifically, enabling the MPC module causes the Program Memory Controller (PMC), Data Memory Controller (DMC), and Unified Memory Controller (UMC) CPU events to be enabled to generate exceptions. The corresponding DMA events are not enabled.

If you want other exceptions to be generated, you need to enable those system events and write functions to handle them. To support this, the EXC module provides the APIs described in this appendix. Since the MPC module takes control of the EXC function hooks, the MPC module also provides a function hook that you can assign to handle additional exception processing (see `_MPC_userHook`).

## C.2.5 Retrieving General Exception Information

The following support routines gather information about the most recent exception or MPC violation:

- ❑ `EXC_getLastStatus`
- ❑ `_MPC_getLastMPFAR`
- ❑ `_MPC_getLastMPFSR`

See Section C.4, *EXC Module* and Section C.5, *\_MPC Module* for details.

### C.3 Data Types and Macros

The following types and macros are defined in `exc.h`, which your application should include if you call EXC APIs.

```
typedef struct EXC_Status {
    Uint32 efr;      /* copy of exception flag register (EFR) */
    Uint32 nrp;      /* copy of NMI return pointer (NRP) */
    Uint32 ntsr;     /* copy of TSR used by NMI processing (NTSR) */
    Uint32 ierr;     /* copy of internal exception report register (IERR) */
} EXC_Status;

/* EFR (Exception Flag Register) bits */
#define EXC_EFRSXF      0x00000001
#define EXC_EFRIXF      0x00000002
#define EXC_EFREXF      0x40000000
#define EXC_EFRNXF      0x80000000

/* ECR (Exception Clear Register) bits */
#define EXC_ECRSXF      EXC_EFRSXF
#define EXC_ECRIXF      EXC_EFRIXF
#define EXC_ECREXF      EXC_EFREXF
#define EXC_ECRNXF      EXC_EFRNXF

/* TSR exception enable bits */
#define EXC_TSRGEE      0x00000004
#define EXC_TSRXEN      0x00000008

/* TSR Privilege Mode bits */
#define EXC_TSRCXMSV    0x00000000
#define EXC_TSRCXMUS    0x00000040

/* IERR (Internal Exception Report Register) bits */
#define EXC_IERRIFX      0x00000001
#define EXC_IERRFPX      0x00000002
#define EXC_IERREPX      0x00000004
#define EXC_IERROPX      0x00000008
#define EXC_IERRRCX      0x00000010
#define EXC_IERRRAX      0x00000020
#define EXC_IERRPRX      0x00000040
#define EXC_IERRLBX      0x00000080
#define EXC_IERRMS      0x00000100

/* MPC CPU Access Memory Protection Fault Events */
#define EXC_EVTMCCMPA    120    /* PMC CPU fault */
#define EXC_EVTDMCCMPA    122    /* DMC CPU fault */
#define EXC_EVTUMCCMPA    124    /* UMC CPU fault */
#define EXC_EVTEMCCMPA    126    /* EMC CPU fault */
```

## C.4 EXC Module

The EXC module supplies the following APIs:

- ❑ EXC\_clearLastStatus. Clears latest exception status values.
- ❑ EXC\_dispatch. Function run by HWI\_NMI.
- ❑ EXC\_evtEvtClear. Clears an event from the event flag register.
- ❑ EXC\_evtExpEnable. Enables an event to generate an exception.
- ❑ EXC\_exceptionHandler. Services non-software exceptions.
- ❑ EXC\_exceptionHook. Hook fxn called by EXC\_exceptionHandler.
- ❑ EXC\_external. Handles exceptions external to the CPU.
- ❑ EXC\_externalHook. Hook fxn called by EXC\_external.
- ❑ EXC\_getLastStatus. Gets latest exception status values.
- ❑ EXC\_internal. Handles exceptions internal to the CPU.
- ❑ EXC\_internalHook. Hook fxn called by EXC\_internal.
- ❑ EXC\_nmi. Handles legacy NMI exceptions.
- ❑ EXC\_nmiHook. Hook fxn called by EXC\_nmi.

### C.4.1 EXC\_dispatch

**Syntax** Void EXC\_dispatch(Void);

**Parameters** None

**Return** None

**Description** When you enable EXC support, the DSP/BIOS HWI\_NMI object is configured to run the EXC\_dispatch function. This function then handles all types of NMIs (non-maskable interrupts). Its actions are determined by the type of NMI that occurs. The types of NMIs are:

- ❑ **Software-generated exceptions.** System calls that generate a SWE instruction are treated as exceptions. EXC\_dispatch calls a function to handle SWE exceptions. Currently, the only case supported is a system call made by MPC\_setPrivMode. The source for this function is provided with the EXC\_dispatch source code; you can modify it to handle additional SWE instructions.
- ❑ **Internal exceptions.** These are routed to EXC\_exceptionHandler, which calls the EXC\_internal API.
- ❑ **External exceptions.** These are MPC violations. These are routed to EXC\_exceptionHandler, which calls the EXC\_external API. If both the MPC and EXC modules are enabled, the \_MPC hook function for external exceptions reports MPC violations.
- ❑ **Legacy NMI.** These are routed to EXC\_exceptionHandler, which calls the EXC\_nmi API.

Note that EXC\_dispatch is not run by the HWI dispatcher and does not use HWI\_enter/HWI\_exit. DSP/BIOS treats exceptions as “dead-end” situations. You can customize the EXC\_exceptionHandler and EXC\_dispatch functions to allow for recovery from exceptions.

Source code for EXC\_dispatch is provided in the exc\_asm.s64P file in the src/exc subdirectory of the DSP/BIOS installation. Source code for other EXC functions is in the exc.c file in the same subdirectory.

**Constraints and  
Calling Context**

- ❑ This function should only be called as the function for the DSP/BIOS HWI\_NMI object.

## C.4.2 EXC\_exceptionHandler

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | Void EXC_exceptionHandler(Void);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Description</b>                     | <p>EXC_exceptionHandler is called by EXC_dispatch to service exceptions that aren't software induced. It performs the following actions:</p> <ol style="list-style-type: none"> <li>1) Reads the EFR (Exception Flag Register) to determine which type of exception to service (internal, external, or legacy NMI).</li> <li>2) Prints the following information about the exception using LOG_error (whose output goes to the "Execution Graph Details" window): <ul style="list-style-type: none"> <li>■ EFR value</li> <li>■ NRP value</li> <li>■ privilege mode (user/supervisor)</li> </ul> </li> <li>3) Records the following information in the EXC_Status structure for later retrieval through a call to EXC_getLastStatus: <ul style="list-style-type: none"> <li>■ EXC_Status.efr</li> <li>■ EXC_Status.nrp</li> <li>■ EXC_Status.ntsrf (contains user/supervisor mode)</li> </ul> </li> <li>4) Calls the hook function pointed to by the EXC_exceptionHook pointer. This hook function must conform to the following prototype: <pre>Void (*EXC_exceptionHook) (Void)</pre> </li> <li>5) Calls the default handler for the type of interrupt. This will be either EXC_internal, EXC_external, or EXC_nmi.</li> <li>6) Terminates the system by calling SYS_abort.</li> </ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"> <li>❑ This function should only be called in the context of exception handling.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

### C.4.3 EXC\_exceptionHook

|                                        |                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void (*EXC_exceptionHook)(Void);</code>                                                                                                                                                                                                                                                                                                                       |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                |
| <b>Description</b>                     | <p>If EXC_exceptionHook points to a function, that function is called by EXC_exceptionHandler after it prints the pertinent exception information and before it calls the default handling function for the NMI type. See EXC_exceptionHandler for further details.</p> <p>For the MPC module, the default EXC_exceptionHook function is _MPC_exceptionHandler.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                         |

### C.4.4 EXC\_internal

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void EXC_internal(Void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b>                     | <p>EXC_internal handles exceptions that are internal to the CPU. That is, they are neither SWE, nor external to the CPU, nor legacy NMI.</p> <p>This function is called by EXC_exceptionHandler. It performs the following actions:</p> <ol style="list-style-type: none"><li>1) Decodes the IERR register and prints the information contained therein using LOG_error.</li><li>2) Calls the hook function pointed to by the EXC_internalHook pointer. This hook function must conform to the following prototype:<br/><br/><code>Void (*EXC_internalHook)(Void)</code></li><li>3) Records the EXC_Status.ierr value for later retrieval and inspection by the user (with a call to EXC_getLastStatus).</li><li>4) Clears the IERR and returns.</li></ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |

### C.4.5 EXC\_internalHook

|                                        |                                                                                                                                                                                                                                                                              |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void (*EXC_internalHook)(Void);</code>                                                                                                                                                                                                                                 |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                         |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                         |
| <b>Description</b>                     | <p>If EXC_internalHook points to a function, that function is called by EXC_internal after it prints information from the IERR register. See EXC_internal for further details.</p> <p>For the MPC module, the default EXC_internalHook function is _MPC_internalHandler.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"> <li>❑ This function should only be called in the context of exception handling.</li> </ul>                                                                                                                                                |

### C.4.6 EXC\_external

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void EXC_external(Void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b>                     | <p>EXC_external handles exceptions that are external to the CPU. That is, they are neither SWE, nor internal to the CPU, nor legacy NMI.</p> <p>This function is called by EXC_exceptionHandler. It performs the following actions:</p> <ol style="list-style-type: none"> <li>1) Prints a message using LOG_error that indicates an external exception occurred.</li> <li>2) Calls the hook function pointed to by the EXC_externalHook pointer. This hook function must conform to the following prototype: <pre>Void (*EXC_externalHook) (Void)</pre> </li> </ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"> <li>❑ This function should only be called in the context of exception handling.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                        |

### C.4.7 EXC\_externalHook

|                                        |                                                                                                                                                                                                                                                                                              |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void (*EXC_externalHook)(Void);</code>                                                                                                                                                                                                                                                 |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                         |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                         |
| <b>Description</b>                     | <p>If EXC_externalHook points to a function, that function is called by EXC_external after it prints information to indicate that an exception occurred. See EXC_external for further details.</p> <p>For the MPC module, the default EXC_externalHook function is _MPC_externalHandler.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                  |

### C.4.8 EXC\_nmi

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void EXC_nmi(Void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>                     | <p>EXC_nmi handles legacy NMI exceptions. That is, they are neither SWE, nor internal or external to the CPU.</p> <p>This function is called by EXC_exceptionHandler. It performs the following actions:</p> <ol style="list-style-type: none"><li>1) Prints a message using LOG_error that indicates a legacy NMI occurred.</li><li>2) Calls the hook function pointed to by the EXC_nmiHook pointer. This hook function must conform to the following prototype:<br/><br/><code>Void (*EXC_nmiHook) (Void)</code></li></ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                   |



### C.4.9 EXC\_nmiHook

|                                        |                                                                                                                                                                                                                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | Void (*EXC_nmiHook)(Void);                                                                                                                                                                                                                                                        |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                              |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                              |
| <b>Description</b>                     | <p>If EXC_nmiHook points to a function, that function is called by EXC_nmi after it prints information to indicate that an exception occurred. See EXC_nmi for further details.</p> <p>For the MPC module, the default EXC_nmiHook function is FXN_F_nop, which does nothing.</p> |
| <b>Constraints and Calling Context</b> | <p>❑ This function should only be called in the context of exception handling.</p>                                                                                                                                                                                                |

### C.4.10 EXC\_getLastStatus

|                    |                                                                                                                                                                                                                                                                                           |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | EXC_Status EXC_getLastStatus(Void);                                                                                                                                                                                                                                                       |
| <b>Parameters</b>  | None                                                                                                                                                                                                                                                                                      |
| <b>Return</b>      | EXC_Status status; /* contains last values of exception registers */                                                                                                                                                                                                                      |
| <b>Description</b> | <p>EXC_getLastStatus retrieves the last recorded values of the exception registers that correspond to the members of the EXC_Status data type. These values are overwritten when the next exception is processed. You can clear the EXC_status fields by calling EXC_clearLastStatus.</p> |

The EXC\_Status data type is as follows:

```
typedef struct EXC_Status {
    Uint32 efr; /* copy of exception flag register (EFR) */
    Uint32 nrp; /* copy of NMI return pointer (NRP) */
    Uint32 ntsr; /* copy of TSR used by NMI processing (NTSR) */
    Uint32 ierr; /* copy of internal exception report register (IERR) */
} EXC_Status;
```

|                                        |                                                                                                                                                                                                                                       |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Constraints and Calling Context</b> | <p>❑ This function is usually called in the context of exception handling. If you extend exception handling support to include recovering from exceptions, this function may be called outside the context of exception handling.</p> |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### C.4.11 EXC\_clearLastStatus

|                    |                                                                                                                                        |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | Void EXC_clearLastStatus(Void);                                                                                                        |
| <b>Parameters</b>  | None                                                                                                                                   |
| <b>Return</b>      | None                                                                                                                                   |
| <b>Description</b> | EXC_clearLastStatus clears the last recorded values of exception registers that correspond to the members of the EXC_Status data type. |

These values will be set to new values when the next exception is processed. They may be retrieved by way of the API EXC\_getLastStatus.

You can use this API along with EXC\_getLastStatus to determine whether a new exception has occurred since the time EXC\_clearLastStatus was called.

#### Constraints and Calling Context

- ❑ This function is usually called in the context of exception handling. If you extend exception handling support to include recovering from exceptions, this function may be called outside the context of exception handling.

### C.4.12 EXC\_evtExpEnable

|                    |                                                                                                                                                                                                                                       |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | Void EXC_evtExpEnable(Uns event);                                                                                                                                                                                                     |
| <b>Parameters</b>  | Uns            event        /* event number */                                                                                                                                                                                        |
| <b>Return</b>      | None                                                                                                                                                                                                                                  |
| <b>Description</b> | EXC_evtExpEnable enables the specified event type to generate the EXCEP hardware exception (which is routed to NMI). You must call this function in order for a particular type of event to be recognized by the exception framework. |

The EXC module provides constants for the following event types. See the “System Event Mapping” table in the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871) for a list of event numbers.

```
/* MPC CPU Access Memory Protection Fault Events */
#define EXC_EVTPMCMPA 120 /* PMC CPU fault */
#define EXC_EVTDMCMPA 122 /* DMC CPU fault */
#define EXC_EVTUMCMPA 124 /* UMC CPU fault */
#define EXC_EVTEMCMPA 126 /* EMC CPU fault */
```

The MPC module enables the first three types of CPU faults as hardware exceptions, but does not enable the EXC\_EVTEMCMPA fault.

#### Constraints and Calling Context

- ❑ none

### C.4.13 EXC\_evtEvtClear

|                                        |                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | Void EXC_evtEvtClear(Uns event);                                                                                                                                                                                                                                                                                                      |
| <b>Parameters</b>                      | Uns            event        /* event number */                                                                                                                                                                                                                                                                                        |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b>                     | <p>EXC_evtEvtClear clears the specified event from the event flag register (EVTFLAGx). It must be called in order for that event to generate a new exception.</p> <p>This function is for external exceptions only. You may want to use it if you write your own EXC_externalHook function. It is called by _MPC_externalHandler.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function is usually called in the context of exception handling. If you extend exception handling support to include recovering from exceptions, this function may be called outside the context of exception handling.</li></ul>                                                        |

## C.5 **\_MPC Module**

The MPC module provides the following handlers, hooks, and functions:

- ❑ `_MPC_exceptionHandler`. Assigned to `EXC_exceptionHook`.
- ❑ `_MPC_getLastMPFAR`. Gets MPFAR for a memory controller.
- ❑ `_MPC_getLastMPFSR`. Gets MPFSR for a memory controller.
- ❑ `_MPC_externalHandler`. Assigned to `EXC_externalHook`.
- ❑ `_MPC_internalHandler`. Assigned to `EXC_internalHook`.
- ❑ `_MPC_userHook`. Hook for user-defined function.

### C.5.1 **\_MPC\_exceptionHandler**

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void _MPC_exceptionHandler(Void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Description</b>                     | <p><code>_MPC_exceptionHandler</code> is assigned to the <code>EXC_exceptionHook</code> function pointer when you enable the MPC module. It performs the following actions:</p> <ol style="list-style-type: none"><li>1) Records exception status (using <code>EXC_getLastStatus</code>) in the structure <code>_MPC_excStatus</code>, which is of type <code>EXC_Status</code>.</li><li>2) Calls the user-settable hook function pointer (<code>_MPC_userHook</code>).</li></ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                                       |

### C.5.2 **\_MPC\_internalHandler**

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | <code>Void _MPC_internalHandler(Void);</code>                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b>                     | <p><code>_MPC_internalHandler</code> is assigned to the <code>EXC_internalHook</code> function pointer when you enable the MPC module. It is a minimal function that only records exception status using <code>EXC_getLastStatus</code>.</p> <p>Typically the MPC module doesn't cause any internal exceptions, but certain MPC exceptions can get flagged as an internal exception when they are caught early by the CPU instead of by an MPC module.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                |

### C.5.3 \_MPC\_externalHandler

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | Void _MPC_externalHandler(Void);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b>                     | <p>_MPC_externalHandler is assigned to the EXC_externalHook function pointer when you enable the MPC module. This is where the bulk of MPC exception processing occurs. This function performs the following actions:</p> <ol style="list-style-type: none"><li>1) Inspects all known MPC controllers for violations and prints any violations using LOG_error.</li><li>2) Records all pertinent information regarding the violation for later retrieval with the _MPC_getLastMPFAR and _MPC_getLastMPFSR APIs.</li><li>3) Clears the event that caused the exception using EXC_evtEvtClear and returns.</li></ol> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

### C.5.4 \_MPC\_userHook

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>                          | Void (*_MPC_userHook)(Void);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Parameters</b>                      | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Return</b>                          | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b>                     | <p>_MPC_userHook is called by _MPC_exceptionHandler. This is a user-settable hook function, so you can replace it with your own function if you like. See _MPC_exceptionHandler for further details.</p> <p>This hook function is called prior to handling the actual MPC violation. If you want the default _MPC_exceptionHandler to handle the violation, don't perturb the existing violation information in the MPC hardware registers.</p> <p>The default _MPC_userHook function is FXN_F_nop, which does nothing.</p> |
| <b>Constraints and Calling Context</b> | <ul style="list-style-type: none"><li>❑ This function should only be called in the context of exception handling.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                 |

### C.5.5 `_MPC_getLastMPFAR`

|                    |                                                                                                                                                                                                                |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <code>UInt32 _MPC_getLastMPFAR(Uns id);</code>                                                                                                                                                                 |
| <b>Parameters</b>  | <code>Uns id /* _MPC_PMC, _MPC_DMC, or _MPC_UMC */</code>                                                                                                                                                      |
| <b>Return</b>      | <code>UInt32 mpfarReg /* Last observed MPFAR register for controller */</code>                                                                                                                                 |
| <b>Description</b> | <code>_MPC_getLastMPFAR</code> returns the latest observed copy of the MPC register MPFAR (Memory Protection Fault Address Register). This register's value is recorded by <code>_MPC_externalHandler</code> . |

Each peripheral that generates memory protection faults provides an MPFAR register. The `id` parameter indicates whether to get the MPFAR for the Program Memory Controller (PMC), Data Memory Controller (DMC), or Unified Memory Controller (UMC). The `_mpc.h` file defines the following constants for use with the `id` parameter:

```
_MPC_DMC
_MPC_PMC
_MPC_UMC
```

#### Constraints and Calling Context

- ❑ This function is usually called during exception handling. If you extend exception handling to include recovering from exceptions, this function may be called outside the context of exception handling.

### C.5.6 `_MPC_getLastMPFSR`

|                    |                                                                                                                                                                                                               |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <code>UInt32 _MPC_getLastMPFSR(Uns id);</code>                                                                                                                                                                |
| <b>Parameters</b>  | <code>Uns id /* _MPC_PMC, _MPC_DMC, or _MPC_UMC */</code>                                                                                                                                                     |
| <b>Return</b>      | <code>UInt32 mpfsrReg /* Last observed MPFSR register for controller */</code>                                                                                                                                |
| <b>Description</b> | <code>_MPC_getLastMPFSR</code> returns the latest observed copy of the MPC register MPFSR (Memory Protection Fault Status Register). This register's value is recorded by <code>_MPC_externalHandler</code> . |

Each peripheral that generates memory protection faults provides an MPFSR register. The `id` parameter indicates whether to get the MPFSR for the Program Memory Controller (PMC), Data Memory Controller (DMC), or Unified Memory Controller (UMC). The `_mpc.h` file defines the following constants for use with the `id` parameter:

```
_MPC_DMC
_MPC_PMC
_MPC_UMC
```

#### Constraints and Calling Context

- ❑ This function is usually called during exception handling. If you extend exception handling to include recovering from exceptions, this function may be called outside the context of exception handling.

# Index

64Plus cache support 2-135

## A

A registers, conventions for B-2  
abort function 2-505  
aborting program 2-506  
allocators  
    for messages sent by MSGQ module 2-274  
    interface for 2-326  
AMR register, conventions for B-3  
AND operation  
    signed integers 2-3  
    unsigned integers 2-4  
Arg data type 1-4  
ArgToInt macro 2-562  
ArgToPtr macro 2-562  
arguments for functions 1-4  
assembly language  
    callable functions (DSP/BIOS) A-2  
    calling C functions from 1-3  
atexit function 2-562  
ATM module 2-2  
    function callability A-2  
    functions in, list of 1-5, 2-2  
ATM\_andi function 2-3  
ATM\_andu function 2-4  
ATM\_cleari function 2-5  
ATM\_clearu function 2-6  
ATM\_deci function 2-7  
ATM\_decu function 2-8  
ATM\_inci function 2-9  
ATM\_incu function 2-10  
ATM\_ori function 2-11  
ATM\_oru function 2-12  
ATM\_seti function 2-13  
ATM\_setu function 2-14  
atomic queue manager 2-388  
average statistics for data series 2-465

## B

B registers, conventions for B-2  
BCACHE module 2-15

    functions in, list of 1-5, 2-15  
BCACHE\_getMar function 2-17  
BCACHE\_getMode function 2-18  
BCACHE\_getSize function 2-19  
BCACHE\_inv function 2-20  
BCACHE\_invL1pAll function 2-21  
BCACHE\_setMar function 2-22  
BCACHE\_setMode function 2-24  
BCACHE\_setSize function 2-25  
BCACHE\_wait function 2-26  
BCACHE\_wb function 2-27  
BCACHE\_wbAll function 2-28  
BCACHE\_wbInv function 2-29  
BCACHE\_wbInvAll function 2-30  
BIOS library  
    instrumented or non-instrumented 2-132  
board clock frequency 2-130  
board input clock 2-137  
board name 2-130  
Bool data type 1-4  
Boolean values 1-4  
BUF module 2-31  
    configuration properties 2-31  
    function callability A-2  
    functions in, list of 1-6, 2-31  
    global properties 2-33  
    object properties 2-33  
BUF\_alloc function 2-35  
BUF\_create function 2-36  
BUF\_delete function 2-38  
BUF\_free function 2-39  
BUF\_maxbuff function 2-40  
BUF\_stat function 2-41  
buffer pool  
    allocating fixed-size buffer 2-35  
    creating 2-36  
    deleting 2-38  
    fixed-size buffers 2-31  
    freeing fixed-size buffer 2-39  
    maximum number of buffers 2-40  
    status of 2-41  
buffered pipe manager 2-306  
buffers, splitting 2-113  
bypass mode 2-18

# C

- C functions
  - calling from assembly language 1-3
- C\_library\_stdlib 2-562
- C62 module 2-42
  - function callability A-3
  - functions in, list of 1-6
- C62\_disableIER function 2-43
- C62\_enableIER function 2-45
- C62\_plug function 2-49
- C64 module 2-42
  - function callability A-3
- C64\_disableIER function 2-44
- C64\_enableIER function 2-47
- C64\_plug function 2-50
- C64x+ interrupt controller 2-118
- c6711 boards
  - memory segments 2-248
- c6x EVM boards
  - memory segments 2-248
- cache
  - invalidate 2-20, 2-21, 2-29, 2-30
  - operations 2-16
  - size 2-19, 2-25
  - writeback 2-27, 2-28, 2-29, 2-30
- cache mode 2-132
- cache support
  - 64Plus 2-135
- cache, L1 and L2 2-15
- callability of functions A-2
- calling context (see context)
- calloc function 2-562
  - not callable from SWI or HWI A-11
- channels (see communication channels; data channels; host channels)
- character, outputting 2-518
- class driver 2-74
- CLK module 2-51
  - checking calling context 2-200
  - configuration properties 2-51
  - function callability 2-176, A-3
  - functions in, list of 1-7, 2-51
  - global properties 2-55
  - object properties 2-59
  - timer for, driving PRD ticks 2-332, 2-333
  - trace types for 2-519
- CLK\_countspms function 2-61
- CLK\_cpuCyclesPerHtime function 2-62
- CLK\_cpuCyclesPerLtime function 2-63
- CLK\_F\_isr function 2-55
- CLK\_gethtime function 2-64
- CLK\_gettime function 2-65
- CLK\_getprd function 2-66
- CLK\_reconfig function 2-67

- CLK\_start function 2-69
- CLK\_stop function 2-70
- clock function
  - not callable from SWI or HWI A-11
- clocks (see clock domains; real-time clock; system clock; timer)
- communication channels
  - closing 2-152
  - control call on 2-149
  - opening 2-150, 2-154
- constraints 2-369
- consumer, of data pipe 2-308
- context
  - CLK, checking for 2-200
  - HWI, checking for 2-200
  - SWI, checking for 2-494
  - switching, functions allowing A-2
  - switching, register usage and 1-3
- conversion specifications for formatted data 2-510, 2-512, 2-514, 2-516
- count statistics for data series 2-465
- counts per millisecond, timer 2-61
- CPU clock domains (see clock domains)
- CPU cycles
  - converting high-resolution time to 2-62
  - converting low-resolution time to 2-63
- CPU frequency 2-138, 2-141
- CPU load 2-356
- CSR register, conventions for B-3

# D

- data channels
  - busy status, checking 2-408
  - initializing 2-409
  - initializing for output 2-410
  - input, disabling 2-411
  - input, enabling 2-413
  - input, number of MADUs read from 2-419
  - input, reading from 2-417, 2-418
  - input, status of 2-415
  - output, disabling 2-412
  - output, enabling 2-414
  - output, status of 2-416
  - output, writing to 2-420
- data pipes 2-306
  - allocating empty frame from 2-312
  - getting frame from 2-315
  - number of frames available to read 2-317
  - number of frames available to write 2-320
  - number of words written, setting 2-325
  - putting frame in 2-323
  - recycling frame that has been read to 2-314
  - writerAddr point of, getting 2-319



- data types 1-4
    - Arg 1-4
    - Bool 1-4
    - EnumInt 1-4
    - EnumString 1-4
    - Extern 1-4
    - Int16 1-4
    - Int32 1-5
    - Numeric 1-5
    - Reference 1-5
    - String 1-5
  - default values
    - for properties 1-4
  - dependencies
    - declaring 2-378
    - number of, determining 2-360
    - releasing 2-376
  - DEV module 2-71
    - configuration properties 2-73
    - function callability A-3
    - functions in, list of 1-7, 2-71
    - object properties 2-74
    - properties 2-74
  - DEV\_createDevice function 2-76
  - DEV\_deleteDevice function 2-79
  - DEV\_match function 2-80
  - device
    - closing 2-81
    - control operation of 2-82
    - creating 2-76
    - deleting 2-79
    - idling 2-83
    - initializing 2-84
    - matching with driver 2-80
    - opening 2-87
    - readiness of, checking 2-88
    - retrieving buffer from 2-89
    - sending buffer to 2-85
  - device drivers 2-71
    - DGN driver 2-91
    - DGS driver 2-95
    - DHL driver 2-99
      - global properties 2-101
      - object properties 2-101
    - DIO adapter 2-103
    - DNL driver 2-106
    - DOV driver 2-107
    - DPI driver 2-109
      - object properties 2-111
    - DST driver 2-113
    - DTR driver 2-115
    - list of 2-74
    - matching device with 2-80
  - device table 2-80
  - device-dependent control operations, performing 2-444
  - DGN driver 2-74, 2-91
    - object properties 2-92
  - DGS driver 2-74, 2-95
  - dgs.h file 2-96
  - DGS\_Params structure 2-95
  - DHL driver 2-74, 2-99
    - global properties 2-101
    - object properties 2-101
  - DIER register, conventions for B-6
  - DIO adapter 2-74, 2-103
    - configuration properties for 2-103
    - global properties 2-104
    - object properties 2-105
  - DMA channel 2-49
  - DNL driver 2-74, 2-106
  - DNUM register, conventions for B-6
  - DOV driver 2-74, 2-107
  - DPI driver 2-74, 2-109
    - object properties 2-111
  - drivers (see device drivers)
  - DSP Endian Mode 2-131
  - DSP speed 2-130
  - DSP/BIOS functions, list of 1-5
  - DSP/BIOS modules, list of 1-2
  - DSP/BIOS version 2-140
  - DST driver 2-74, 2-113
  - DTR driver 2-74, 2-115
  - dtr.h file 2-116
  - DTR\_multiply function 2-115
  - DTR\_multiplyInt16 function 2-115
  - DTR\_Params structure 2-116
  - Dxx\_close function 2-81
  - Dxx\_ctrl function 2-82
  - Dxx\_idle function 2-83
  - Dxx\_init function 2-84
  - Dxx\_issue function 2-85
  - Dxx\_open function 2-87
  - Dxx\_ready function 2-88
  - Dxx\_reclaim function 2-89
- ## E
- ECM module 2-118
    - configuration properties 2-118
    - functions in, list of 1-8, 2-118
    - global properties 2-120
    - object properties 2-120
  - ECM\_disableEvent function 2-122
  - ECM\_dispatch function 2-123
  - ECM\_dispatchPlub function 2-125
  - ECM\_enableEvent function 2-126
  - ECR register, conventions for B-5
  - EFR register, conventions for B-5
  - empty devices 2-106
  - endian mode 2-131
  - enumerated integers 1-4

- enumerated strings 1-4
- EnumInt data type 1-4
- EnumString data type 1-4
- environment for HOOK and TSK objects 2-166
- environment pointer for HOOK and TSK objects 2-167
- error condition
  - flagging 2-508
- error function 2-505
- error handling
  - error codes A-12
  - MSGQ module 2-301
- error message, writing to system log 2-219
- error number for tasks 2-545
- event combiner 2-118
  - configuring 2-125
  - handling functions 2-123
- events
  - mapping 2-195
  - power, function to be called on 2-371, 2-374
  - scheduling functions based on 2-332
  - tracing 2-519
  - unregistering notification function for 2-387
- EXC module C-2, C-7
  - functions in, list of 1-8
- EXC\_clearLastStatus function C-14
- EXC\_dispatch function C-8
- EXC\_evtEvtClear function C-15
- EXC\_evtExpEnable function C-14
- EXC\_exceptionHandler function C-9
- EXC\_exceptionHook function C-10
- EXC\_external function C-11
- EXC\_externalHook function C-12
- EXC\_getLastStatus function C-13
- EXC\_internal function C-10
- EXC\_internalHook function C-11
- EXC\_nmi function C-12
- EXC\_nmiHook function C-13
- exception handling C-2
- exit function 2-505, 2-562
- exit handler
  - stacking 2-507
- Extern data type 1-4

## F

- f32toi16 function 2-97
- FADCR register, conventions for B-3
- false/true values 1-4
- fatal error C-2
- FAUCR register, conventions for B-4
- fixed-size buffers
  - allocating 2-35
  - freeing 2-39

- maximum number of 2-40
- pools of 2-31
- FMCR register, conventions for B-4
- formatted data, outputting 2-510, 2-512, 2-514, 2-516
- fprintf function
  - not callable from SWI or HWI A-11
- frame
  - available to read to, getting number of 2-317
  - available to write, getting number of 2-320
  - getting from pipe 2-315
  - number of words in, getting 2-318
  - number of words that can be written to 2-321
  - putting in pipe 2-323
  - recycling 2-314
  - size and address of, determining 2-322
- free function 2-562
  - not callable from SWI or HWI A-11
- freeze mode 2-18
- frequency
  - changing 2-347
  - for setpoint, determining 2-365
- functions
  - arguments for 1-4
  - callability of A-2
  - calling conventions for 1-3
  - external 1-4
  - list of 1-5
  - naming conventions for 1-3

## G

- gather/scatter driver 2-95
- GBL module 2-127
  - configuration properties 2-127
  - function callability A-4
  - functions in, list of 1-9, 2-127
  - global properties 2-130
- GBL\_getClkin function 2-137
- GBL\_getFrequency function 2-138
- GBL\_getProcId function 2-139
- GBL\_getVersion function 2-140
- GBL\_setFrequency function 2-141
- GBL\_setProcId function 2-142
- Gconf
  - underscore preceding C function names 1-3, 2-60, 2-204, 2-478
- generators 2-91
- getenv function 2-562
  - not callable from SWI or HWI A-11
- GFPGFR register, conventions for B-4
- GIO module 2-143
  - configuration properties 2-145
  - function callability A-4

- functions in, list of 1-9, 2-143
- global properties 2-146
- object properties 2-147
- GIO\_abort function 2-148
- GIO\_control function 2-149
- GIO\_create function 2-150
- GIO\_delete function 2-152
- GIO\_flush function 2-153
- GIO\_new function 2-154
- GIO\_read function 2-156
- GIO\_submit function 2-158
- GIO\_write function 2-160
- global settings 2-127
- GPLYA register, conventions for B-5
- GPLYB register, conventions for B-5

## H

- hardware interrupts 2-173
  - callable functions A-2
  - context of, determining if in 2-200
  - disabled, manipulating variables while 2-2
  - disabling 2-185
  - enabling 2-190
  - plugging dispatcher 2-187
  - restoring context before interrupt 2-196
  - restoring global interrupt enable state 2-201
  - saving context of 2-192
  - saving or restoring registers across B-2
  - target-specific, disabling 2-43, 2-44
  - target-specific, enabling 2-45, 2-47
  - target-specific, enabling and disabling 2-42
- hardware registers
  - MEM module and 2-239
- hardware timer counter register ticks 2-51
- heap, address 2-254
- high-resolution time 2-51, 2-52, 2-54
  - converting to CPU cycles 2-62
  - getting 2-64
- hook functions 2-162
- HOOK module 2-162
  - configuration properties 2-162
  - function callability A-4
  - functions in, list of 1-10, 2-162
  - object properties 2-164
  - properties 2-164
- HOOK\_getenv function 2-166
- HOOK\_setenv function 2-167
- host channel manager 2-168
- host link driver 2-74, 2-99
- HST module 2-168
  - configuration properties 2-168
  - function callability A-4
  - functions in, list of 1-10, 2-168

- global properties 2-169
- object properties 2-170
- HST object 2-99
- HST\_getpipe function 2-172
- HWI module 2-173
  - configuration properties 2-173
  - function callability A-4
  - functions in, list of 1-10, 2-173
  - global properties 2-178
  - object properties 2-178
  - statistics units for 2-465
- HWI\_applyWugenMasks function 2-184
- HWI\_disable function 2-185
- HWI\_disableWugen function 2-186
- HWI\_dispatchplug function 2-187
- HWI\_enable function 2-190
- HWI\_enableWugen function 2-191
- HWI\_enter function 2-176, 2-192
- HWI\_eventMap function 2-195
- HWI\_exit function 2-176, 2-196
- HWI\_getWugenMasks function 2-198
- HWI\_ierToWugenMasks function 2-199
- HWI\_isHWI function 2-200
- HWI\_NMI object C-8
- HWI\_restore function 2-201

## I

- I/O availability, scheduling functions based on 2-332
- i16tof32 function 2-97
- i16toi32 function 2-97
- i16tou8 function 2-97
- i32toi16 function 2-97
- ICR register, conventions for B-3
- IDL module 2-202
  - configuration properties 2-202
  - function callability A-5
  - functions in, list of 1-10, 2-202
  - global properties 2-203
  - object properties 2-204
- IDL\_run function 2-205
- idle functions, running 2-205
- idle thread manager 2-202
- IER (Interrupt Enable Register)
  - disable interrupts using 2-43, 2-44
  - enable interrupts using 2-45
- IER register, conventions for B-3
- IERR register C-2
- IERR register, conventions for B-5
- IFR register, conventions for B-3
- ILC register, conventions for B-5
- initialization 2-162
- input channels
  - declaring 2-409

- disabling 2-411
- enabling 2-413
- number of MADUs read from 2-419
- reading from 2-417, 2-418
- status of, determining 2-415
- input streams 2-434
- Input/Output
  - aborting 2-148
  - closing communication channel 2-152
  - control call on communication channel 2-149
  - flushing input and output channels 2-153
  - opening communication channel 2-150, 2-154
  - submitting GIO packet 2-158
  - synchronous read 2-156
  - synchronous write 2-160
- INT status bit B-5
- Int16 data type 1-4
- Int32 data type 1-5
- integers
  - enumerated 1-4
  - unsigned 1-5
- interface for allocators 2-326
- interrupt controller 2-118
- Interrupt Enable Register
  - disable interrupts using 2-43, 2-44
  - enable interrupts using 2-45
- interrupt selection number 2-119
- interrupt selector 2-118
- Interrupt Service Fetch Packet 2-49, 2-50
- interrupt service routines (see hardware interrupts)
- Interrupt Service Table 2-49, 2-50
- interrupt threads 2-477
- interrupt vector, plugging 2-42, 2-49, 2-50
- invalidate cache 2-16
- IOM model for device drivers 2-71
- IRP register, conventions for B-3
- ISFP (Interrupt Service Fetch Packet) 2-49, 2-50
- ISR epilog 2-196
- ISR prolog 2-192
- ISR register, conventions for B-3
- ISRs
  - disabling 2-122
  - enabling 2-126
  - handling 2-123
- IST (Interrupt Service Table) 2-49, 2-50
- ISTP register, conventions for B-3
- ITSR register, conventions for B-5

## L

- L1 cache 2-15
- L1D cache 2-16, 2-135
- L1P cache 2-16, 2-135
- L2 cache 2-15, 2-16, 2-135

- L2 cache mode 2-133
- L2 memory settings 2-132, 2-133
- L2 priority queues 2-134
- L2 requestor priority 2-134
- L2 transfer requests 2-134
- L2ALLOC queues 2-134
- latency to scale between setpoints 2-367
- LCK module 2-206
  - configuration properties 2-206
  - function callability A-5
  - functions in, list of 1-11, 2-206
  - global properties 2-206
  - object properties 2-207
- LCK\_create function 2-208
- LCK\_delete function 2-209
- LCK\_pend function 2-210
  - thread restrictions for 2-563
- LCK\_post function 2-212
  - thread restrictions for 2-563
- load addresses 2-244
- load, CPU 2-356, 2-362, 2-377
- localcopy function 2-97
- LOG module 2-213
  - configuration properties 2-213
  - function callability A-5
  - functions in, list of 1-11, 2-213
  - global properties 2-214
  - object properties 2-215
- LOG\_disable function 2-217
- LOG\_enable function 2-218
- LOG\_error function 2-219
- LOG\_event function 2-220
- LOG\_event5 function 2-221
- LOG\_message function 2-223
- LOG\_printf function 2-224
- LOG\_printf4 function 2-227
- LOG\_reset function 2-229
- low-resolution time 2-51, 2-52, 2-53
  - converting to CPU cycles 2-63
  - getting 2-65
  - restarting 2-69
  - stopping 2-70

## M

- MADUs 2-238
- mailbox
  - clear bits from 2-480, 2-482
  - creating 2-232
  - decrementing 2-485
  - deleting 2-233
  - get value of 2-491
  - incrementing 2-493
  - OR mask with value in 2-495, 2-496

- posting message to 2-235
- waiting for message from 2-234
- mailbox manager 2-230
- main function
  - calling context 2-200
- malloc function 2-562
  - not callable from SWI or HWI A-11
- MAR register
  - getting 2-17, 2-22
- MAR registers 2-133, 2-135
- maskable CPU interrupts 2-118
- maximum statistics for data series 2-465
- MBX module 2-230
  - configuration properties 2-230
  - function callability A-5
  - functions in, list of 1-11, 2-230
  - global properties 2-231
  - object properties 2-231
- MBX\_create function 2-232
- MBX\_delete function 2-233
- MBX\_pend function 2-234
- MBX\_post function 2-235
- MEM module 2-236
  - configuration properties 2-236
  - function callability A-5
  - functions in, list of 1-12, 2-236
  - global properties 2-239
  - object properties 2-246
- MEM\_alloc function 2-249
- MEM\_define function 2-251
- MEM\_free function 2-253
- MEM\_getBaseAddress function 2-254
- MEM\_increaseTableSize function 2-255
- MEM\_redefine function 2-256
- MEM\_stat function 2-257
- MEM\_undefine function 2-258
- MEM\_valloc function 2-259
- memory block
  - freeing 2-253
  - increasing 2-255
- memory segment manager 2-236
- memory segments
  - allocating and initializing 2-259
  - allocating from 2-249
  - c6711 boards 2-248
  - c6x EVM boards 2-248
  - defining 2-251
  - existing, redefining 2-256
  - status of, returning 2-257
  - undefining 2-258
- message log 2-213
  - appending formatted message to 2-224, 2-227
  - disabling 2-217
  - enabling 2-218
  - resetting 2-229
  - writing unformatted message to 2-220, 2-221
- message queues 2-273
  - closing 2-281
  - determining destination queue for message 2-286
  - finding 2-293
  - number of messages in 2-282
  - open, finding 2-291
  - opening 2-295
  - placing message in 2-298
  - receiving message from 2-284
  - releasing 2-300
- messages
  - allocating 2-280
  - determining destination message queue of 2-286
  - freeing 2-283
  - ID for, setting 2-303
  - ID of, determining 2-287
  - number of, in message queue 2-282
  - placing in message queue 2-298
  - receiving from message queue 2-284
  - reply destination of, determining 2-289
  - reply destination of, setting 2-305
  - size of, determining 2-288
- messaging, multi-processor 2-270
- mini-drivers 2-103
  - deleting 2-152
- minit function
  - not callable from SWI or HWI A-11
- modules
  - \_MPC module C-16
  - ATM module 2-2
  - BCACHE module 2-15
  - BUF module 2-31
  - C62 module 2-42
  - C64 module 2-42
  - CLK module 2-51
  - DEV module 2-71
  - ECM module 2-118
  - EXC module C-7
  - functions for, list of 1-5
  - GBL module 2-127
  - GIO module 2-143
  - HOOK module 2-162
  - HST module 2-168
  - HWI module 2-173
  - IDL module 2-202
  - LCK module 2-206
  - list of 1-2
  - LOG module 2-213
  - MBX module 2-230
  - MEM module 2-236
  - MPC module 2-260
  - MSGQ module 2-270
  - PIP module 2-306
  - POOL module 2-326

- PRD module 2-331
- PWRM module 2-339
- QUE module 2-388
- SEM module 2-421
- SIO module 2-434
- STS module 2-464
- SWI module 2-474
- SYS module 2-503
- trace types for 2-519
- TRC module 2-519
- TSK module 2-524
- \_MPC module C-16
  - functions in, list of 1-12
- MPC module 2-260
  - configuration properties 2-260
  - exception handling C-5
  - functions in, list of 1-12, 2-260
  - global properties 2-262
- \_MPC\_exceptionHandler function C-16
- \_MPC\_externalHandler function C-17
- \_MPC\_getLastMPFAR function C-18
- \_MPC\_getLastMPFSR function C-18
- MPC\_getPA function 2-263
- MPC\_getPageSize function 2-264
- MPC\_getPrivMode function 2-265, 2-269
- \_MPC\_internalHandler function C-16
- MPC\_setBufferPA function 2-266
- MPC\_setPA function 2-268
- \_MPC\_userHook function C-17
- MSGQ API 2-273, 2-274
- MSGQ module 2-270
  - configuration properties 2-272
  - function callability A-6
  - functions in, list of 1-13, 2-270
  - global properties 2-279
  - internal errors, handling 2-301
  - static configuration 2-275
- MSGQ\_alloc function 2-280
- MSGQ\_close function 2-281
- MSGQ\_count function 2-282
- MSGQ\_free function 2-283
- MSGQ\_get function 2-284
- MSGQ\_getAttrs function 2-285
- MSGQ\_getDstQueue function 2-286
- MSGQ\_getMsgId function 2-287
- MSGQ\_getMsgSize function 2-288
- MSGQ\_getSrcQueue function 2-289
- MSGQ\_isLocalQueue function 2-290
- MSGQ\_locate function 2-291
- MSGQ\_locateAsync function 2-293
- MSGQ\_open function 2-295
- MSGQ\_put function 2-298
- MSGQ\_release function 2-300
- MSGQ\_setErrorHandler function 2-301
- MSGQ\_setMsgId function 2-303

- MSGQ\_setSrcQueue function 2-305
- multiple processors 2-142
- multiprocessor application
  - converting single-processor application to 2-111
- multi-processor applications 2-142
- multi-processor messaging 2-270

## N

- naming conventions
  - functions 1-3
  - properties 1-4
- NMI exceptions C-3
- NMI functions
  - calling HWI functions 2-176
- notification function signatures 2-374
- notification functions 2-387
- notifyReader function 2-308
  - PIP API calls and 2-177
- notifyWriter function 2-308
- NRP register, conventions for B-3
- NTSR register, conventions for B-5
- null driver 2-106
- Numeric data type 1-5

## O

- object references
  - properties holding 1-5
- on-chip timer (see timer)
- operations (see functions)
- OR operation
  - signed integers 2-11
  - unsigned integers 2-12
- output channels
  - declaring 2-410
  - disabling 2-412
  - enabling 2-414
  - status of, determining 2-416
  - writing to 2-420
- output streams 2-434
- outputting formatted data 2-510, 2-512, 2-514, 2-516
- outputting single character 2-518
- overlap driver 2-107

## P

- packing/unpacking ratio, DGS driver 2-95
- PCE1 register, conventions for B-3
- period register
  - value of 2-66
- periodic function
  - starting 2-336

- stopping 2-337
- periodic function manager 2-331
- periodic rate 2-52
- PIP module 2-306
  - configuration properties 2-307
  - function callability A-6
  - functions in, list of 1-14, 2-306
  - global properties 2-309
  - object properties 2-309
  - statistics units for 2-465
  - trace types for 2-519
- PIP\_alloc function 2-312
- PIP\_free function 2-308, 2-314
- PIP\_get function 2-315
- PIP\_getReaderAddr function 2-316
- PIP\_getReaderNumFrames function 2-317
- PIP\_getReaderSize function 2-318
- PIP\_getWriterAddr function 2-319
- PIP\_getWriterNumFrames function 2-320
- PIP\_getWriterSize function 2-321
- PIP\_peek function 2-322
- PIP\_put function 2-308, 2-323
- PIP\_setWriterSize function 2-325
- pipe driver 2-74, 2-109
- pipe manager, buffered 2-306
- pipe object 2-172
- pipes
  - allocating empty frame from 2-312
  - get readerAddr pointer of 2-316
  - getting frame from 2-315
  - number of frames available to read 2-317
  - number of frames available to write 2-320
  - number of words written, setting 2-325
  - putting frame in 2-323
  - recycling frame that has been read to 2-314
  - writerAddr point of, getting 2-319
- POOL module 2-326
  - configuration properties 2-326
  - functions in, list of 2-326
  - global properties 2-330
- power event
  - function to be called on 2-371
  - registered, function to be called on 2-374
- power management 2-339
- PRD module 2-331
  - configuration properties 2-331
  - function callability A-7
  - functions in, list of 1-14, 2-331
  - global properties 2-332
  - object properties 2-333
  - statistics units for 2-465
  - ticks driven by CLK timer 2-332, 2-333
  - ticks, getting current count 2-335
  - ticks, incrementing 2-338
  - ticks, setting increments for 2-333
  - trace types for 2-519
- PRD\_getticks function 2-335
- PRD\_start function 2-336
- PRD\_stop function 2-337
- PRD\_tick function 2-338
- prescaler register
  - resetting 2-67
- printf function
  - not callable from SWI or HWI A-11
- processor ID 2-130, 2-139, 2-142
- processors
  - multiple 2-142
- PROCID 2-142
- producer, of data pipe 2-308
- program
  - aborting 2-506
  - terminating 2-509
- properties
  - data types for 1-4
  - default values for 1-4
  - ECM object 2-120
  - GIO object 2-147
  - HOOK module 2-164
  - HOOK object 2-164
  - MEM object 2-246
  - naming conventions 1-4
- putc function 2-505
- PWRM module 2-339
  - capabilities of, determining 2-352
  - configuration properties 2-341, 2-350
  - constraints, determining 2-354
  - function callability A-7
  - functions in, list of 1-14, 2-339
  - global properties 2-342
- PWRM\_changeSetpoint function 2-347
- PWRM\_configure function 2-350
- PWRM\_getCapabilities function 2-352
- PWRM\_getConstraintInfo function 2-354
- PWRM\_getCPULoad function 2-356
- PWRM\_getCurrentSetpoint function 2-358
- PWRM\_getDependencyCount function 2-360
- PWRM\_getLoadMonitorInfo function 2-362
- PWRM\_getNumSetpoints function 2-363
- PWRM\_getSetpointInfo function 2-365
- PWRM\_getTransitionLatency function 2-367
- PWRM\_registerConstraint function 2-369
- PWRM\_registerNotify function 2-371
- PWRM\_releaseDependency function 2-376
- PWRM\_resetCPULoadHistory 2-377
- PWRM\_setDependency function 2-378
- PWRM\_signalEvent 2-379
- PWRM\_sleepDSP function 2-381
- PWRM\_startCPULoadMonitoring 2-384
- PWRM\_stopCPULoadMonitoring 2-385
- PWRM\_unregisterContrain 2-386

PWRM\_unregisterNotify function 2-387  
 pwrMNotifyFxn function 2-374

## Q

QUE module 2-388  
   configuration properties 2-388  
   function callability A-7  
   functions in, list of 1-15, 2-388  
   global properties 2-389  
   object properties 2-390  
 QUE\_create function 2-391  
 QUE\_delete function 2-392  
 QUE\_dequeue function 2-393  
 QUE\_empty function 2-394  
 QUE\_enqueue function 2-395  
 QUE\_get function 2-396  
 QUE\_head function 2-397  
 QUE\_insert function 2-398  
 QUE\_new function 2-399  
 QUE\_next function 2-400  
 QUE\_prev function 2-401  
 QUE\_put function 2-402  
 QUE\_remove function 2-403  
 queue manager 2-388  
 queues  
   creating 2-391  
   deleting 2-392  
   emptying 2-399  
   getting element from front of 2-396  
   inserting element at end of 2-395  
   inserting element in middle of 2-398  
   putting element at end of 2-402  
   removing element from front of 2-393  
   removing element from middle of 2-403  
   returning pointer to element at front of 2-397  
   returning pointer to next element of 2-400  
   returning pointer to previous element of 2-401  
   testing if empty 2-394

## R

rand function  
   not callable from SWI or HWI A-11  
 reader, of data pipe 2-308  
 readers, MSGQ module 2-272, 2-274  
 read-time data exchange settings 2-405  
 realloc function 2-562  
   not callable from SWI or HWI A-11  
 real-time clock (see CLK module)  
 Reference data type 1-5  
 register conventions B-2  
 registers  
   modification in multi-threaded application B-2

  saving or restoring across function calls or interrupts B-2  
 resource lock  
   acquiring ownership of 2-210  
   creating 2-208  
   deleting 2-209  
   relinquishing ownership of 2-212  
 resource lock manager 2-206  
 resources  
   declaring dependency on 2-378  
   number of dependencies on 2-360  
   releasing dependency on 2-376  
 RILC register, conventions for B-5  
 RTDX module 2-405  
   configuration properties 2-405  
   function callability A-8  
   functions in, list of 1-16  
   object properties 2-407  
   target configuration properties 2-406  
 RTDX\_channelBusy function 2-408  
 RTDX\_CreateInputChannel 2-409  
 RTDX\_CreateOutputChannel function 2-410  
 RTDX\_disableInput function 2-411  
 RTDX\_disableOutput function 2-412  
 RTDX\_enableInput function 2-413  
 RTDX\_enableOutput function 2-414  
 RTDX\_isInputEnabled function 2-415  
 RTDX\_isOutputEnabled function 2-416  
 RTDX\_read function 2-417  
 RTDX\_readNB function 2-418  
 RTDX\_sizeofInput function 2-419  
 RTDX\_write function 2-420  
 RTS functions  
   not calling in HWI or SWI threads 2-175, 2-563  
 RTS library 2-130

## S

scaling operation 2-115  
 SEM module 2-421  
   configuration properties 2-421  
   function callability A-8  
   functions in, list of 1-17, 2-421  
   global properties 2-423  
   object properties 2-423  
 SEM\_count function 2-424  
 SEM\_create function 2-425  
 SEM\_delete function 2-426  
 SEM\_new function 2-427  
 SEM\_pend function 2-428  
 SEM\_pendBinary function 2-429  
 SEM\_post function 2-431  
 SEM\_postBinary function 2-432  
 SEM\_reset 2-433



- semaphore manager 2-421
- semaphores
  - binary, signaling 2-432
  - binary, waiting for 2-429
  - count of, determining 2-424
  - count of, resetting 2-433
  - creating 2-425
  - deleting 2-426
  - initializing 2-427
  - signaling 2-431
  - waiting for 2-428
- setpoints (see V/F setpoints)
- signal generators 2-91
- signed integers
  - AND operation 2-3
  - clearing 2-5
  - decrementing 2-7
  - incrementing 2-9
  - OR operation 2-11
  - setting 2-13
- single-processor application
  - converting to multiprocessor application 2-111
- SIO module 2-434
  - configuration properties 2-435
  - function callability A-8
  - functions in, list of 1-17
  - functions in, list of 2-434
  - global properties 2-436
  - object properties 2-436
- SIO/DEV model for device drivers 2-72
- SIO\_bufsize function 2-440
- SIO\_create function 2-441
- SIO\_ctrl function 2-444
- SIO\_delete function 2-445
- SIO\_flush function 2-446
- SIO\_get function 2-447
- SIO\_idle function 2-449
- SIO\_issue function 2-450
- SIO\_ISSUERECLAIM streaming model
  - DPI and 2-110
- SIO\_put function 2-452
- SIO\_ready function 2-454
- SIO\_reclaim function 2-455
- SIO\_reclaimx function 2-458
- SIO\_segid function 2-459
- SIO\_select function 2-460
- SIO\_staticbuf function 2-462
- sleep
  - changing sleep states 2-381
  - for tasks 2-557
- software generator driver 2-74
- software interrupt manager 2-474
- software interrupts
  - address of currently executing interrupt 2-500
  - attributes of, returning 2-489
  - attributes of, setting 2-501
  - callable functions A-2
  - checking to see if in context of 2-494
  - clearing 2-483
  - context of, determining if in 2-494
  - deleting 2-486
  - disabled, manipulating variables while 2-2
  - enabling 2-488
  - mailbox for, clearing bits 2-480, 2-482
  - mailbox for, decrementing 2-485
  - mailbox for, incrementing 2-493
  - mailbox for, OR mask with value in 2-495, 2-496
  - mailbox for, returning value of 2-491
  - posting 2-495, 2-496, 2-497
  - priority mask, returning 2-492
  - raising priority of 2-498
  - restoring priority of 2-499
- split driver 2-113
- sprintf function
  - not callable from SWI or HWI A-11
- srand function
  - not callable from SWI or HWI A-11
- SSR register, conventions for B-5
- stack
  - allocating for tasks 2-531
  - checking for overflow 2-534
- stack size for tasks 2-529, 2-531
- stackable gather/scatter driver 2-95
- stackable overlap driver 2-107
- stackable split driver 2-113
- stackable streaming transformer driver 2-115
- STATICPOOL allocator 2-328
- statistics
  - resetting values of 2-471
  - saving values for delta 2-472
  - tracing 2-519
  - updating 2-469
  - updating with delta 2-470
- statistics object manager 2-464
- status bit TSR.INT B-5
- std.h library
  - functions in 2-562
  - macros in, list of 1-20
- stdlib.h library
  - functions in 2-562
  - functions in, list of 1-20
- stream I/O manager 2-434
- streams
  - acquiring static buffer from 2-462
  - closing 2-445
  - device for, determining if ready 2-454
  - device for, selecting ready device 2-460
  - device-dependent control operation, issuing 2-444
  - flushing 2-446
  - getting buffer from 2-447

- idling 2-449
  - memory segment used by, returning 2-459
  - opening 2-441
  - putting buffer to 2-452
  - requesting buffer from 2-455, 2-458
  - sending buffer to 2-450
  - size of buffers used by, determining 2-440
  - strftime function
    - not callable from SWI or HWI A-11
  - String data type 1-5
  - strings 1-5
    - enumerated 1-4
  - STS module 2-464
    - configuration properties 2-464
    - function callability A-9
    - functions in, list of 1-18, 2-464
    - global properties 2-467
    - object properties 2-467
  - STS\_add function 2-469
  - STS\_delta function 2-470
  - STS\_reset function 2-471
  - STS\_set function 2-472
  - sum statistics for data series 2-465
  - SWE instruction C-2
  - SWI module 2-474
    - configuration properties 2-475
    - function callability A-9
    - functions in, list of 1-18, 2-474
    - global properties 2-478
    - object properties 2-478
    - statistics units for 2-465
    - trace types for 2-519
  - SWI\_andn function 2-480
  - SWI\_andnHook function 2-482
  - SWI\_create function 2-483
  - SWI\_dec function 2-485
  - SWI\_delete function 2-486
  - SWI\_enable function 2-488
  - SWI\_getattr function 2-489
  - SWI\_getmbx function 2-491
  - SWI\_getpri function 2-492
  - SWI\_inc function 2-493
  - SWI\_isSWI function 2-494
  - SWI\_or function 2-495
  - SWI\_orHook function 2-496
  - SWI\_post function 2-497
  - SWI\_raisepri function 2-498
  - SWI\_restorepri function 2-499
  - SWI\_self function 2-500
  - SWI\_setattr function 2-501
  - synchronous read 2-156
  - synchronous write 2-160
  - SYS module 2-503
    - configuration properties 2-503
    - function callability A-9
    - functions in, list of 1-19, 2-503
    - global properties 2-504
    - object properties 2-505
  - SYS\_abort function 2-505, 2-506
  - SYS\_atexit function 2-507
  - SYS\_EALLOC status A-12
  - SYS\_EBADIO status A-12
  - SYS\_EBADOBJ status A-12
  - SYS\_EBUSY status A-12
  - SYS\_EDEAD status A-12
  - SYS\_EDOMAIN status A-12
  - SYS\_EEOF status A-12
  - SYS\_EFREE status A-12
  - SYS\_EINVAL status A-12
  - SYS\_EMODE status A-12
  - SYS\_ENODEV status A-12
  - SYS\_ENOTFOUND status A-12
  - SYS\_ENOTIMPL status A-12
  - SYS\_error function 2-505, 2-508
  - SYS\_ETIMEOUT status A-12
  - SYS\_EUSER status A-12
  - SYS\_exit function 2-505, 2-509
  - SYS\_OK status A-12
  - SYS\_printf function 2-505, 2-510
  - SYS\_putchar function 2-505, 2-518
  - SYS\_sprintf function 2-512
  - SYS\_vprintf function 2-505, 2-514
  - SYS\_vsprintf 2-516
  - system clock 2-52
    - choosing module driving 2-530
    - incrementing in TSK module 2-550, 2-559
    - PRD module driving 2-530
    - returning current value of 2-560
  - system clock manager 2-51
  - system events
    - exceptions C-2
  - system log 2-213
    - writing error message to 2-219
    - writing program-supplied message to 2-223
  - system settings, managing 2-503
- ## T
- target board name 2-130
  - task environment
    - setting 2-552
  - task manager 2-524
  - task scheduler
    - disabling 2-541
    - enabling 2-542
  - tasks
    - callable functions A-2
    - checking if in context of 2-549
    - creating 2-535

- currently executing, handle of 2-551
  - default priority of 2-529
  - delaying execution of (sleeping) 2-557
  - deleting 2-538
  - environment pointer for, getting 2-544
  - error number for, getting 2-545
  - error number for, setting 2-553
  - execution priority of, setting 2-554
  - handle of STS object, getting 2-548
  - incrementing system clock for 2-550, 2-559
  - name of, getting 2-546
  - not shutting down system during 2-533
  - priority of 2-532, 2-547
  - resetting time statistics for 2-555
  - status of, retrieving 2-558
  - terminating 2-543
  - updating time statistics for 2-539
  - yielding to task of equal priority 2-561
  - Tconf
    - underscore preceding C function names 1-3, 2-60, 2-204, 2-478
  - terminating program 2-509
  - threads
    - idle thread manager 2-202
    - interrupt threads 2-477
    - register modification and B-2
    - RTS functions callable from 2-563
  - tick count, determining 2-335
  - tick counter (see PRD module, ticks)
  - timer 2-51, 2-52
    - counts per millisecond 2-61
    - resetting 2-67
    - specifying 2-54
  - timer counter 2-53
  - timer mode 2-54
  - timer period register
    - resetting 2-67
  - timestamps 2-215
  - trace buffer
    - memory segment for 2-504
    - size of 2-504
  - trace manager 2-519
  - tracing
    - disabling 2-521
    - enabling 2-522
    - querying enabled trace types 2-523
  - transform function, DGS driver 2-95
  - transformer driver 2-115
  - transformers 2-115
  - transports array 2-142, 2-278
  - transports, MSGQ module 2-274
  - TRC module 2-519
    - function callability A-10
    - functions in, list of 1-19, 2-519
  - TRC\_disable function 2-521
  - TRC\_enable function 2-522
  - TRC\_query function 2-523
  - true/false values 1-4
  - TSCH register, conventions for B-6
  - TSCL register, conventions for B-6
  - TSK module 2-524
    - configuration properties 2-525
    - function callability A-10
    - functions in, list of 1-19, 2-524
    - global properties 2-529
    - object properties 2-531
    - statistics units for 2-465
    - system clock driven by 2-530, 2-550, 2-559
    - trace types for 2-519
  - TSK\_checkstacks function 2-534
  - TSK\_create function 2-535
  - TSK\_delete function 2-538
  - TSK\_deltatime function 2-539
  - TSK\_disable function 2-541
  - TSK\_enable function 2-542
  - TSK\_exit function 2-543
  - TSK\_getenv function 2-544
  - TSK\_geterr function 2-545
  - TSK\_getname function 2-546
  - TSK\_getpri function 2-547
  - TSK\_getsts function 2-548
  - TSK\_isTSK function 2-549
  - TSK\_itick function 2-550
  - TSK\_self function 2-551
  - TSK\_setenv function 2-552
  - TSK\_seterr function 2-553
  - TSK\_setpri function 2-554
  - TSK\_settime function 2-555
  - TSK\_sleep function 2-557
  - TSK\_stat function 2-558
  - TSK\_tick function 2-559
  - TSK\_time function 2-560
  - TSK\_yield function 2-561
  - TSR register, conventions for B-5
  - TSR.INT status bit B-5
- ## U
- u16tou32 function 2-97
  - u32tou16 function 2-97
  - u32tou8 function 2-97
  - u8toi16 function 2-97
  - u8tou32 function 2-97
  - underscore
    - preceding C function names 1-3, 2-60, 2-204, 2-478
  - unsigned integers 1-5
    - AND operation 2-4
    - clearing 2-6

- decrementing 2-8
- incrementing 2-10
- OR operation 2-12
- setting 2-14

## V

- V/F setpoints
  - changing 2-347
  - determining 2-358
  - frequency and voltage of, determining 2-365
  - latency to scale between 2-367
  - number of determining 2-363
- variables
  - manipulating with interrupts disabled 2-2
- vectID parameter 2-195
- vfprintf function

- not callable from SWI or HWI A-11
- voltage
  - changing 2-347
  - for setpoint, determining 2-365
- fprintf function
  - not callable from SWI or HWI A-11
- vsprintf function
  - not callable from SWI or HWI A-11

## W

- wait for cache operation 2-26
- writeback cache 2-16
- writer, of data pipe 2-308
- writers, MSGQ module 2-272, 2-275
- WUGEN registers 2-184, 2-186, 2-191, 2-198, 2-199